

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



# DISSERTATION

**A PIPELINED VECTOR PROCESSOR  
AND MEMORY ARCHITECTURE FOR  
CYCLOSTATIONARY PROCESSING**

by

Raymond F. Bernstein, Jr.

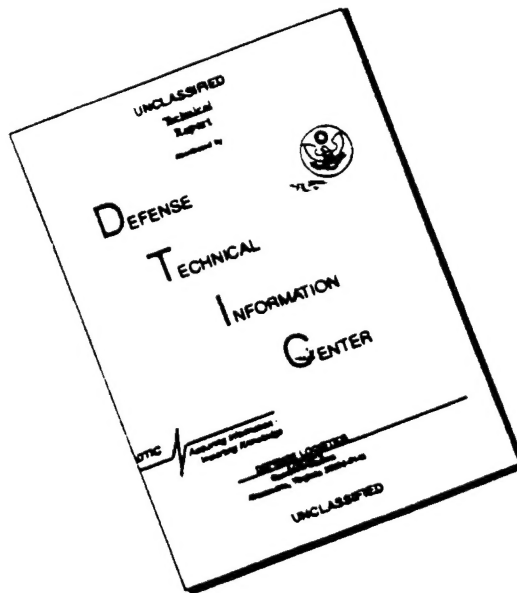
Dissertation Supervisor:

Herschel H. Loomis, Jr.

Approved for public release; distribution is unlimited.

19960328 047

# DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1995	3. REPORT TYPE AND DATES COVERED Doctoral Dissertation	
4. TITLE AND SUBTITLE A Piplined Vector Processor and Memory Architecture for Cyclostationary Processing			5. FUNDING NUMBERS	
6. AUTHOR(S) Raymond F. Bernstein, Jr.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This work describes a scaleable, high performance, pipelined, vector processor architecture. Special emphasis is placed on performing fast Fourier transforms with mixed-radix butterfly operations. The initial motivation for the architecture was the computation of cyclostationary algorithms. However, the resulting architecture is capable of general purpose vector processing as well. A major factor affecting the performance of the architecture is the memory system design. The use of pipelining techniques, coupled with vector processing, places a substantial burden on the memory system performance. The memory design is based on an interleaved memory philosophy with a buffering technique referred to as split transaction memory (STM). A crucial aspect of the memory design is the memory decoding scheme. A design methodology is described for the specification of permutation matrices that yield near optimal performance for the memory system. Another important aspect of this work is the development of a software based simulator that allows a STM to be specified. The simulator, operating at the register transfer level, emulates the processing of an address stream by STM and records the events for post-processing. The STM simulator was used to evaluate three types of vector processing address patterns: constant stride, constant geometry radix-r butterfly, and digit reversed. A random address pattern was also analyzed in the context of general-purpose computing. STM simulation verified the near optimal performance of the STM.				
14. SUBJECT TERMS computer architecture, pipelined vector processing, interleaved memory, fast Fourier transform, permutation matrix			15. NUMBER OF PAGES 275	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	





Approved for public release; distribution unlimited.

# A PIPELINED VECTOR PROCESSOR AND MEMORY ARCHITECTURE FOR CYCLOSTATIONARY PROCESSING

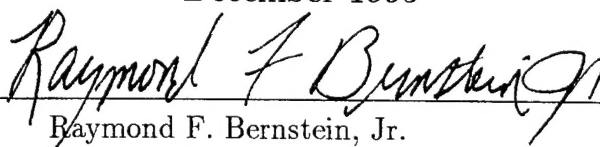
Raymond F. Bernstein, Jr.  
B.S., Texas Tech University, 1977  
M.S., Naval Postgraduate School, 1982

## DOCTOR OF PHILOSOPHY IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL  
December 1995

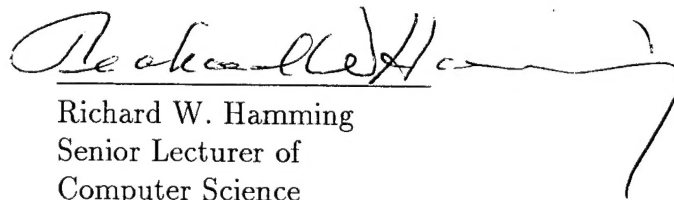
Author: \_\_\_\_\_

  
Raymond F. Bernstein, Jr.

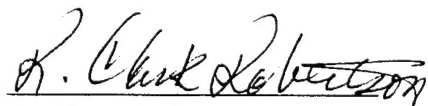
Approved by: \_\_\_\_\_



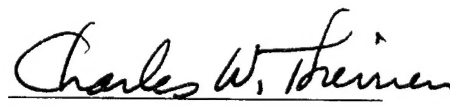
Douglas J. Fouts  
Asst. Professor of Electrical  
and Computer Engineering



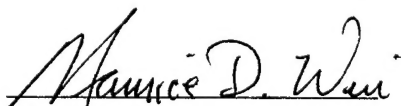
Richard W. Hamming  
Senior Lecturer of  
Computer Science



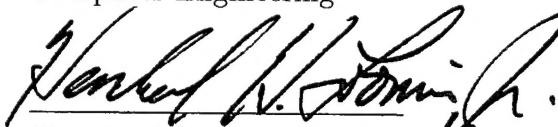
R. Clark Robertson  
Assoc. Professor of Electrical  
and Computer Engineering



Charles W. Therrien  
Professor of Electrical and  
Computer Engineering

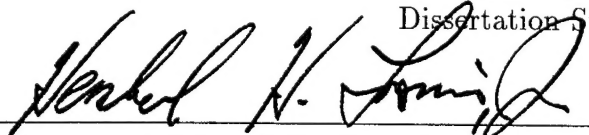


Maurice D. Weir  
Professor of Mathematics



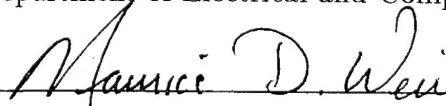
Herschel H. Loomis, Jr.  
Professor of Electrical and  
Computer Engineering  
and Space Systems,  
Dissertation Supervisor

Approved by: \_\_\_\_\_



Herschel H. Loomis, Jr., Chairman  
Department of Electrical and Computer Engineering

Approved by: \_\_\_\_\_



Maurice D. Weir, Associate Provost for Instruction



## ABSTRACT

This work describes a scaleable, high-performance, pipelined, vector processor architecture. Special emphasis is placed on performing fast Fourier transforms with mixed-radix butterfly operations. The initial motivation for the architecture was the computation of cyclostationary algorithms. However, the resulting architecture is capable of general-purpose vector processing as well. A major factor affecting the performance of the architecture is the memory system design. The use of pipelining techniques, coupled with vector processing, places a substantial burden on the memory system performance. The memory design is based on an interleaved memory philosophy with a buffering technique referred to as split transaction memory (STM). A crucial aspect of the memory design is the memory decoding scheme. A design methodology is described for the specification of permutation matrices that yield near-optimal performance for the memory system. Another important aspect of this work is the development of a software based simulator that allows a STM to be specified. The simulator, operating at the register transfer level, emulates the processing of an address stream by STM and records the events for post-processing. The STM simulator was used to evaluate three types of vector processing address patterns: constant stride, constant geometry radix- $r$  butterfly, and digit reversed. A random address pattern was also analyzed in the context of general-purpose computing. STM simulation verified the near-optimal performance of the STM.



## TABLE OF CONTENTS

<b>I. INTRODUCTION .....</b>	<b>1</b>
A. BACKGROUND.....	1
B. VECTOR ARCHITECTURE.....	3
C. PROBLEM STATEMENT.....	5
D. ORGANIZATION OF DISSERTATION .....	7
E. ORIGINAL CONTRIBUTION.....	8
<b>II. HISTORICAL PERSPECTIVE AND RELATED RESEARCH.....</b>	<b>11</b>
A. THE GENERAL PROBLEM.....	11
B. MEMORY ADDRESS STREAM.....	14
C. CACHE MEMORY.....	18
D. INTERLEAVED MEMORY .....	22
<b>III. BUTTERFLY MACHINE ARCHITECTURE .....</b>	<b>41</b>
A. INTRODUCTION .....	41
B. BASIC ARCHITECTURAL CONCEPTS.....	43
C. PERFORMANCE MEASURES .....	50
D. FAST FOURIER TRANSFORM.....	51
E. PERMUTATION-BASED MEMORY DECODING SCHEME .....	55
F. ONE-CHIP ARCHITECTURE.....	67
G. PARALLEL ARCHITECTURE .....	71
<b>IV. DESCRIPTION OF SPLIT TRANSACTION MEMORY .....</b>	<b>75</b>
A. PHYSICAL DESCRIPTION.....	75
B. SIMULATION MODEL .....	85
1. Signal Generators .....	85
2. STM Simulator.....	88
3. Graphics Programs .....	93
<b>V. THEORETICAL PERFORMANCE ANALYSIS OF STM .....</b>	<b>99</b>

A. CONSTANT STRIDE.....	99
B. RADIX-R BUTTERFLY ADDRESSING.....	107
C. DIGIT REVERSAL.....	111
D. PERMUTATION-BASED DECODING PERFORMANCE.....	114
E. RANDOM ADDRESSING .....	124
<b>VI. SIMULATION STUDIES .....</b>	<b>127</b>
A. OVERVIEW.....	127
B. VECTOR PROCESSING EXPERIMENTS .....	147
1. Constant Stride: Conventional Memory Decoding .....	147
2. Constant Stride: Permutation-Based Memory Decoding .....	162
3. Radix-r Butterfly: Conventional Memory Decoding.....	177
4. Radix-r Butterfly: Permutation-Based Memory Decoding .....	187
5. Digit Reversed: Conventional Memory Decoding .....	200
6. Digit Reversed: Permutation-Based Memory Decoding .....	204
C. GENERAL-PURPOSE COMPUTING EXPERIMENT.....	210
<b>VII. CONCLUSIONS .....</b>	<b>213</b>
A. DESIGN DECISIONS.....	213
B. STM DESIGN METHODOLOGY .....	215
C. GENERAL CONCLUSIONS.....	216
<b>LIST OF REFERENCES .....</b>	<b>219</b>
<b>APPENDIX MATLAB SOURCE CODE FOR STM SIMULATOR.....</b>	<b>223</b>
<b>INITIAL DISTRIBUTION LIST .....</b>	<b>253</b>

## LIST OF FIGURES

I.1 Spectral Correlation Function for BPSK Signal .....	2
II.1 Generic Vector Processor Architecture .....	13
II.2 General-Purpose Processor .....	15
II.3 Vector Processor .....	17
II.4 Cache Memory System .....	19
II.5 Interleaved Memory Block Diagram .....	23
II.6 Cray I Memory Hierarchy .....	28
II.7 Interleaved Memory With Queues .....	34
II.8 Split Transaction Memory Overview.....	37
II.9 Comparison of Buffers Versus Cache Elements.....	39
III.1 Butterfly Machine Environment.....	42
III.2 General Vector Machine Architecture.....	44
III.3 Vector Timing Diagram.....	45
III.4 1024-Point FFT: Pass 1 .....	48
III.5 1024-Point FFT: Pass 2 .....	48
III.6 1024-Point FFT: Pass 3 .....	49
III.7 Timing Characteristics for 1024-Point FFT .....	49
III.8 Radix-2 In-place Decimation-in-frequency Flow Graph.....	52
III.9 Radix-2 Constant-Geometry Decimation-in-frequency Flow Graph .....	52
III.10 Radix-4/Radix-2 In-place Decimation-in-frequency Flow Graph.....	54
III.11 Permutation Address Pattern Maps .....	65
III.12 Comparison of Permutation Address Patterns.....	66
III.13 Simulation Permutation Matrix: NoBanks=4.....	66
III.14 Simulation Permutation Matrix: NoBanks=8.....	66
III.15 Simulation Permutation Matrix: NoBanks=16.....	67
III.16 Simulation Permutation Matrix: NoBanks=32.....	67
III.17 One-Chip Architecture .....	68
III.18 SSCA Functional Diagram .....	68
III.19 SSCA Execution: Channelization .....	69

III.20	SSCA Execution: Correlation Multiply .....	70
III.21	SSCA Execution: N FFT .....	70
III.22	Parallel Architecture (One Board) .....	72
III.23	Process Allocation: Ratio of Backend to Channelizer Cycles .....	73
III.24	Processing Efficiency for SSCA (Ten Processor System) .....	74
IV.1	Split Transaction Memory Overview .....	76
IV.2	Cache Element .....	77
IV.3	Top Level Memory System .....	79
IV.4	Smart Cache Design .....	81
IV.5	Relationship Between Smart Cache Counters .....	82
IV.6	Block A for Figure V-4 .....	84
IV.7	STM Simulation Overview .....	86
IV.8	Simplified Algorithmic Description of stm .....	92
IV.9	Example Plot From m_anal Function .....	96
IV.10	Example Mesh Plot for the Performance Parameter Speedup .....	97
V.1	Interleaved Memory Address Space: Conventional Bank Selection .....	100
V.2	Timing Diagram: Optimal Throughput .....	103
V.3	Timing Diagram: Non-Optimal Throughput (cont.) .....	106
V.4	Radix-2 Constant Geometry Decimation-in-Frequency FFT .....	108
V.5	Timing Diagram for Radix-4 Butterfly Pattern (Standard Interleaving) .....	109
V.6	Timing Diagram for Radix-4 Butterfly Pattern STM(4,5,4) .....	111
V.7	Required Mappings When the Stride Equals the Number of Banks .....	118
V.8	Mapping Required When Stride is One Half the Number of Banks .....	121
VI.1	Steady-State Throughput for Strides=1,3,5,7,9 (Conventional Decoding) .....	130
VI.2	Maximum Latency for Strides=1,3,5,7,9 (Conventional Decoding) .....	130
VI.3	Steady-State Throughput for Stride=2, 6 (Conventional Decoding) .....	131
VI.4	Maximum Latency for Stride=2, 6 (Conventional Decoding) .....	131
VI.5	Steady-State Throughput for Stride=4 (Conventional Decoding) .....	132
VI.6	Maximum Latency for Stride=4 (Conventional Decoding) .....	132
VI.7	Steady-State Throughput for Stride=8 (Conventional Decoding) .....	133



VI.8 Maximum Latency for Stride=8 (Conventional Decoding) .....	133
VI.9 Steady-State Throughput for Stride= $2^k$ for $k = 0,1,2 \dots$ (Permutation-Based Decoding) .....	135
VI.10 Maximum Latency for Stride= $2^k$ for $k = 0,1,2 \dots$ (Permutation-Based Decoding) .....	135
VI.11 Steady-State Throughput for Radix=2 (Conventional Decoding).....	137
VI.12 Maximum Latency for Radix=2 (Conventional Decoding) .....	137
VI.13 Maximum Latency for Radix=4 (Conventional Decoding) .....	138
VI.14 Maximum Latency for Radix=8 (Conventional Decoding) .....	138
VI.15 Maximum Latency for Radix=16 (Conventional Decoding) .....	139
VI.16 Steady-State Throughput for Radix=2, 4, 8, and 16 (Permutation-Based Decoding) .....	140
VI.17 Maximum Latency for Radix=2, 4, 8, and 16 (Permutation-Based Decoding) .....	140
VI.18 Steady-State Throughput for Radix=2 (Conventional Decoding).....	141
VI.19 Steady-State Throughput for Radix=2/NoDigits=10 (Permutation-Based Decoding) .....	143
VI.20 Maximum Latency for Radix=2/NoDigits=10 (Permutation-Based Decoding) .....	143
VI.21 Steady-State Throughput for Radix=4/NoDigits=5 (Permutation-Based Decoding) .....	144
VI.22 Maximum Latency for Radix=4/NoDigits=5 (Permutation-Based Decoding) .....	144
VI.23 Steady-State Throughput for Radix=8/NoDigits=4 (Permutation-Based Decoding) .....	145
VI.24 Maximum Latency for Radix=8/NoDigits=4 (Permutation-Based Decoding) .....	145
VI.25 Steady-State Throughput for Radix=16/NoDigits=3 (Permutation-Based Decoding) .....	146
VI.26 Maximum Latency for Radix=16/NoDigits=3 ..	

(Permutation-Based Decoding) .....	146
VI.27 Comparison of Theoretical Versus Simulated Steady-State Throughput for Strides=1,3,5,7,9 (Conventional Decoding) .....	149
VI.28 Comparison of Theoretical Versus Simulated Maximum Latency for Strides=1,3,5,7,9 (Conventional Decoding).....	150
VI.29 Detailed Simulation Run for Stride=1 STM(4,2,4).....	151
VI.30 Detailed Simulation Run for Stride=1 STM(32,2,32).....	152
VI.31 Average Simulated Throughput for Stride=1 (Conventional Decoding) .....	153
VI.32 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=2 (Conventional Decoding).....	154
VI.33 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=2 (Conventional Decoding).....	155
VI.34 Detail Simulation Run for Stride=2 STM(32,3,32) (Conventional Decoding) .....	156
VI.35 Average Simulated Throughput for Stride=2 (Conventional Decoding) .....	157
VI.36 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=4 (Conventional Decoding).....	158
VI.37 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=4 (Conventional Decoding).....	159
VI.38 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=8 (Conventional Decoding).....	160
VI.39 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=8 (Conventional Decoding).....	161
VI.40 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=1 (Permutation-Based Decoding).....	165
VI.41 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=1 (Permutation-Based Decoding).....	166
VI.42 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=2 (Permutation-Based Decoding).....	167
VI.43 Comparison of Theoretical Versus Simulated Maximum Latency	

for Stride=2 (Permutation-Based Decoding).....	168
VI.44 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Stride=64 (Permutation-Based Decoding).....	169
VI.45 Comparison of Theoretical Versus Simulated Maximum Latency	
for Stride=64 (Permutation-Based Decoding).....	170
VI.46 Detailed Simulation Run for Stride=64 STM(4,3,4)	
(Permutation-Based Decoding) .....	171
VI.47 Detail Simulation Run for Stride=64 STM(32,3,32)	
(Permutation-Based Decoding) .....	172
VI.48 Second Detailed Simulation Run for Stride=64 STM(32,3,32)	
(Permutation-Based Decoding) .....	173
VI.49 Simulated Average Throughput for Stride=64	
(Permutation-Based Decoding) .....	174
VI.50 Simulated Steady-State Throughput and Average Throughput for	
Stride=3 (Permutation-Based Decoding) .....	175
VI.51 Simulated Steady-State Throughput and Average Throughput for	
Stride=5 (Permutation-Based Decoding) .....	176
VI.52 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=2 (Conventional Decoding).....	178
VI.53 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=2 (Conventional Decoding).....	179
VI.54 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=4 (Conventional Decoding).....	180
VI.55 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=4 (Conventional Decoding).....	181
VI.56 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=8 (Conventional Decoding).....	182
VI.57 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=8 (Conventional Decoding).....	183
VI.58 Comparison of Theoretical Versus Simulated Steady-State Throughput	

for Radix=16 (Conventional Decoding).....	184
VI.59 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=16 (Conventional Decoding).....	185
VI.60 Average Throughput for Radix-2 Butterfly Pattern (Conventional Decoding)....	186
VI.61 Average Throughput for Radix-16 Butterfly Pattern	
(Conventional Decoding) .....	187
VI.62 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=2 (Permutation-Based Decoding).....	189
VI.63 Comparison of Theoretical Versus Simulated Maximum Latency for	
Radix=2 (Permutation-Based Decoding) .....	190
VI.64 Detail Simulation Run for Radix-2 STM(8,3,8)	
(Permutation-Based Decoding) .....	191
VI.65 Detail Simulation Run for Radix-2 STM(8,4,8)	
(Permutation-Based Decoding) .....	192
VI.66 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=4 (Permutation-Based Decoding).....	194
VI.67 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=4 (Permutation-Based Decoding).....	195
VI.68 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=8 (Permutation-Based Decoding).....	196
VI.69 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=8 (Permutation-Based Decoding).....	197
VI.70 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=16 (Permutation-Based Decoding).....	198
VI.71 Comparison of Theoretical Versus Simulated Maximum Latency	
for Radix=16 (Permutation-Based Decoding).....	199
VI.72 Comparison of Theoretical Versus Simulated Steady-State Throughput	
for Radix=2 / NoDigits=10 (Conventional Decoding).....	201
VI.73 Detail Simulation Run for Radix=2 / NoDigits=10 STM(4,3,4)	
(Conventional Decoding) .....	202

VI.74	Detail Simulation Run for Radix=2 / NoDigits=10 STM(32,3,32)	
	(Conventional Decoding) .....	203
VI.75	Comparison of Theoretical Versus Simulated Steady-State Throughput	
	for Radix=2 / NoDigits=10 (Permutation-Based Decoding).....	205
VI.76	Comparison of Theoretical Versus Simulated Maximum Latency for	
	Radix=2 / NoDigits=10 (Permutation-Based Decoding) .....	206
VI.77	Comparison of Theoretical Versus Simulated Steady-State Throughput	
	for Radix=4 / NoDigits=5 (Permutation-Based Decoding).....	207
VI.78	Comparison of Theoretical Versus Simulated Steady-State Throughput	
	for Radix=8 / NoDigits=4 (Permutation-Based Decoding).....	208
VI.79	Comparison of Theoretical Versus Simulated Steady-State Throughput	
	for Radix=16 / NoDigits=3 (Permutation-Based Decoding).....	209
VI.80	General-Purpose Experiment: Speedup .....	211
VI.81	General-Purpose Experiment: Throughput .....	211
VI.82	General-Purpose Experiment: Maximum Latency .....	212



## LIST OF TABLES

III.1 Pass Definition.....	46
III.2 Pass Description .....	47
III.3 Binary Counting Sequence .....	61
IV.1 Digit Reversal for Three Digits Base 2 .....	87
VI.1 Vector Processor Experiments .....	128
VI.2 NoCE Evaluated in the Third Vector Processor Experiment.....	136
VI.3 NoCE Evaluated in the Sixth Vector Processor Experiment .....	142
VI.4 General-Purpose Computer Experiment .....	147
VI.5 Comparison of Analytic Versus Simulated Speedup .....	210





## **ACKNOWLEDGMENT**

I give thanks to my committee for their dedication to my education. A very special thank you to my advisor, Professor Herschel. H. Loomis Jr. His support and guidance have been unfailing. This gift of knowledge is one which I can never repay. This endeavor would not have been possible without his long term commitment.

As much credit must go to my wife and friend Anna Lee, as to me for her patience and endurance over the past six years. Thank you also for bringing into this life Lauren, Paul, and Claire during this time period.

All praise and glory to my Lord and Savior Jesus Christ who makes all things possible.



## I. INTRODUCTION

### A. BACKGROUND

This research began with an investigation of computer architectures for computing digital implementations of the *Spectral Correlation Function* (SCF), the central function of spectral correlation or cyclostationary analysis. The SCF is defined as:

$$S_{X_T}^{\alpha}(t, f)_{\Delta t} = \frac{1}{\Delta t} \int_{t-\Delta t/2}^{t+\Delta t/2} \frac{1}{T} X_T(w, f + \frac{\alpha}{2}) X_T^*(w, f - \frac{\alpha}{2}) dw \quad (I.1)$$

where

$$X_T(t, f) = \int_{t-T/2}^{t+T/2} x(w) e^{-j2\pi fw} dw. \quad [\text{Ref 1}] \quad (I.2)$$

and  $T$  is the length of a time window for Equation (I.2). The variable  $f$  is called the spectral location parameter and corresponds to the frequency parameter of a Fourier transform pair. It is expressed as

$$f = \frac{1}{2} \left[ \left( f + \frac{\alpha}{2} \right) + \left( f - \frac{\alpha}{2} \right) \right]. \quad (I.3)$$

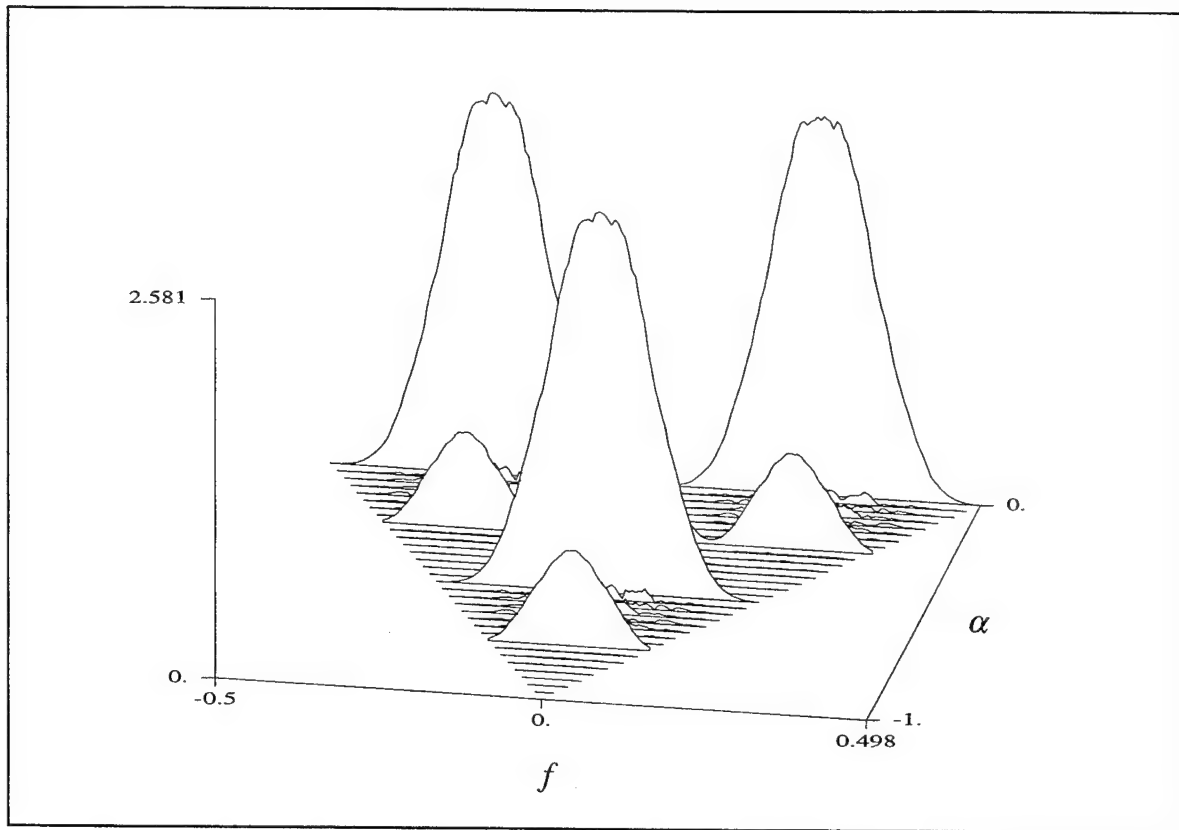
$\alpha$  is the spectral separation parameter representing a frequency of second-order periodicity.  $\alpha$ , also referred to as the cycle frequency, is expressed as

$$\alpha = \left( f + \frac{\alpha}{2} \right) - \left( f - \frac{\alpha}{2} \right). \quad (I.4)$$

$X_T(t, f)$  is the Fourier transform of the time series signal  $x(t)$  of length  $T$  centered at time  $t$ .

The SCF of most man-made signals result in non-zero cycle frequencies. An example of a magnitude plot of a SCF for a binary phase shift keyed (BPSK) signal is shown in Figure I.1. Each non-zero line (called a *feature*) in the plot corresponds to a unique value of  $\alpha$ . The traditional power spectral density is a special case of spectral correlation analysis (i.e., the line for  $\alpha = 0$ ). The power spectral density is the feature at the back of the plot. Three smaller cycle features as well as a large cycle feature can be seen in the plot. Unlike the power spectral density, noise present in other cycle

frequencies will not correlate, and with sufficient averaging, will yield a feature regardless of the noise level. This provides a means for detecting a weak signal hidden in noise.



**Figure I.1 Spectral Correlation Function for BPSK Signal**

Several digital algorithms have been developed for estimating the SCF including the *Frequency Smoothing Method* (FSM) and the time smoothed variants *FFT Accumulation Method* (FAM) and the *Strip Spectral Correlation Algorithm* (SSCA) [Ref 2] [Ref 3]. Each of these algorithms are heavily based on vector processing in general and FFT techniques in particular. The computational complexity of these algorithms has been extensively analyzed. Applications for cyclostationary analysis can be found in Gardner [Ref 1] and Gardner [Ref 4]. The computational complexity for the SSCA will be discussed further in Chapter 0, Section F. A computer designed to exploit spectral correlation features is referred to as a spectral correlation analyzer (SCA).

A variety of architectures were investigated for computing the SCF including networks of general-purpose computers, digital signal processing (DSP) architectures, and more specialized architectures based on vector processing techniques.

For example, the SSCA was implemented on a network of Sun workstations connected with an Ethernet. The software used to facilitate communications and control of the distributed application was Parallel Virtual Machine (PVM). It was found that the SSCA could be partitioned in such a way to permit effective parallel execution on numerous workstations. This provides a means for computing a computationally intensive instance of the SSCA during off peak hours of the computing facilities.

The primary focus however was to find computer architectures that would compute the SCF in real or near-real time. Transputers were examined to determine feasibility of real time and near-real time computation for the *fast Fourier transforms* (FFTs) in particular and spectral correlation algorithms in general. The Transputer is a general-purpose processor that contains support for quick context switching and communications on chip. It is designed to be scaleable and is a valid technology for many application domains. However, the number of Transputers that would be required to provide the needed computation was found to be too many for a reasonable implementation for this application.

Highly specialized architectures have also been considered for several cyclostationary algorithms. Architectures for both frequency and time smoothing algorithms may be found in Roberts [Ref 5] and [Ref 6]. These architectures are based on mapping hardware onto the algorithmic requirements thereby providing architectures that can yield optimal performance. A practical disadvantage to this approach is the reduced cost effectiveness of a hardware implementation that is dedicated to a particular algorithm.

## **B. VECTOR ARCHITECTURE**

Another architecture reviewed was based on *vector array processors*. This approach, the subject of this dissertation, is based on streaming data through a highly pipelined vector processor with, in the ideal case, no wait states. The basic concept can

be used to build highly optimized architectures for many but not all of the functions needed in a SCA (i.e., those portions that can be vectorized). Alternatively, this basic approach can be used to build a more generalized vector processor that might be used for any problem that lends itself to vector processing. This more generalized approach will be referred to hereafter as the *butterfly machine architecture*. The butterfly machine (BFM) architecture can also be scaled. An architecture designed with multiple vector processors will also be described and is referred to as the *parallel butterfly machine architecture*. This name is not to be confused with the BBN Butterfly by BBN Advanced Computers [Ref 7].

Given the technology available today, the key issue to consider when evaluating the butterfly machine architecture is the requirements of the memory system. The streaming of data through the pipelined vector processor requires a data reference from each vector each clock cycle. A typical vector operation requires two input vectors and creates one output vector, therefore implying three data references per clock cycle per processor. Given that the vector processor is pipelined, the clock rate applied to the processor will be on the higher end of the scale available with current technology. Multiple memory references per clock cycle and a high clock rate suggest that designing a memory to accommodate this requirement is a primary area of concern.

As will be seen in Chapter 0, the butterfly machine architecture calls for several large memories for each vector processor. Given the data rate requirements stated above, such a memory system could be accommodated by using fast *static random access memory* (SRAM). This is not a desirable solution because SRAMs are much more expensive per bit relative to the *dynamic random access memory* (DRAM) alternative. Secondary factors favoring a bulk storage approach such as DRAM include their need for less power and circuit board real-estate. The issue of cost becomes more acute when considering the parallel butterfly machine architecture. Therefore, a cost effective implementation of the butterfly machine architecture will use bulk storage technology such as DRAM instead of SRAM given the current technology base.

DRAM has been the memory technology for implementing main memory in general-purpose computers for some time. Almost any general-purpose computer acquired today will have a memory system that is composed of a main memory consisting of DRAM technology coupled with one or two levels of SRAM-based *cache* memory. However, vector array machines frequently rely on some form of *banked interleaved* memory (i.e., a memory system consisting of parallel memories that attempts to exploit the parallelism to increase throughput). The relative merits of cache versus interleaved memory techniques for a memory system will be discussed in detail in Chapter II.

### C. PROBLEM STATEMENT

This dissertation describes a computer architecture that is optimized for vector processing in general and cyclostationary processing in particular. The memory system design is the key component of this architecture for the reasons discussed in Section B above.

There are two characteristics of the butterfly machine environment that distinguish it from a general-purpose computing environment and have a substantial effect on the solution to the memory system. First, the memory references are very dense when compared to the general-purpose computing case. As indicated in the discussion above, a data reference is required for each vector on each cycle. This imposes a requirement of the memory system that is more stringent than would be expected for a general-purpose computer.

The second characteristic of the butterfly machine environment is that all memory addresses are known before the first elements of the vector are processed. Therefore, a memory reference stream can be generated with certainty for instructions and data to be executed in the future. This implies that substantial latency can be tolerated given that the vector length is long relative to the latency. Note that this is in sharp contrast to the general-purpose computing architecture where very little latency can be tolerated without having a substantial impact on performance. It will be shown how this latency is traded for memory bandwidth using interleaved memory.

Two aspects of the butterfly machine architecture diminish the usefulness of traditional caching techniques. First, since memory must operate at the same speed as the processor, there are no “processing only” cycles that can be used for loading the cache in parallel. This becomes increasingly more important when the size of the cache lines are large with respect to the bus size. Additionally, address reference patterns associated with vector processing often do not meet the locality of reference criteria needed for a memory system using a cache.

The primary objective of this dissertation is to define a low-cost *attached vector processor* architecture that is well suited for cyclostationary analysis. In particular, this architecture will perform fast Fourier transforms (FFTs) and other vector operations to include complex addition and multiplication. By low cost, it is meant that the vector processor architecture is compatible with workstations rather than mainframes or supercomputers. A key component of this architecture is a memory system that incorporates low-cost bulk storage memory. Although DRAMs, the current choice given today’s technology continue to increase in capability, their access speeds are slower than microprocessors by as much as a factor of ten or more. This research addresses the design of a memory system, based primarily on relatively slow bulk storage devices, that will provide memory bandwidth that is sufficient to maintain optimum processor performance. The technique used to construct such a memory is referred to as *Split Transaction Memory* (STM).

STM will also be analyzed in the context of general-purpose computing. This investigation into general-purpose computing provides a more comprehensive understanding of the use of STM for other computing environments.

Architectures for computing FFTs have been studied since the late sixties. One of the earliest works is by Pease [Ref 8]. A hardwired signal processor for radar applications is described by Groginsky [Ref 9]. Another developed by Lincoln Labs Massachusetts Institute of Technology, is found in Filip [Ref 10]. This processor is designed using multiple microprocessors that communicate via a bus. Two methods for resolving the bit-reversal problem are discussed by Dieffenderfer [Ref 11]. Another



hardwired processor, using a radix-4 butterfly, is presented by Corinthios [Ref 12]. This processor supports real-time applications transforming 256-point vectors with signal sampling rates up to 1.6 million samples a second. A VLSI architecture is proposed by Sapiecha [Ref 13]. This architecture consists of two and three dimensional arrays of processor elements. Two real-time processors include a Winograd Fourier transform processor presented by Sommer [Ref 14] and a processor designed for synthetic-aperture-radar applications Franceschetti [Ref 15].

The work contained in this document is distinguished from the work noted above in that the processor is designed for general vector processing as well as FFT computation. The architecture presented in this dissertation is particularly well suited for input vectors of length  $2^{20}$  and larger. The application is scaleable providing a real-time or near-real-time response. Major emphasis is on a low-cost memory design.

## **D. ORGANIZATION OF DISSERTATION**

The following conventions will be used to more clearly identify features of the document. Regular text is in Times New Roman font. Computer program names, algorithms, and variables are printed with Arial font. Other variables discussed in a different context than a program are shown as *italic Times New Roman*.

The remainder of this dissertation is organized as follows. Chapter II, Historical Perspective and Related Research, provides a brief description and comparison of several computer architectures and a comparison of cache and interleaving memory schemes. A history of related research in interleaved memory is then presented.

The next chapter, Butterfly Machine Architecture, describes the butterfly machine architecture and provides a context for use of STM.

Chapter IV, Description of Split Transaction Memory (STM), presents STM first at a conceptual level, followed by a hardware design. A description of the STM Simulator is then presented.

A theoretical model of STM performance parameters is described in Chapter V, first using conventional bank number decoding, followed by permutation-based decoding.

Chapter VI, Simulation Studies, describes the experiments. The theoretical performance of the STM, based on the results of Chapter V, is detailed for each experiment. The results of the simulation runs of each experiment is described and compared to the theoretical performance. Conclusions that are specific to an experiment are also stated.

Chapter VII lists top level conclusions and describes further research.

The following section describes the original contributions of this work.

## **E. ORIGINAL CONTRIBUTION**

The primary contribution of this research is an attached vector processor architecture designed for executing algorithms that require an efficient implementation of vector processing in general, and the fast Fourier transform (FFT) in particular. Classes of problems addressed by this type of architecture include signal processing, spectral analysis, digital filtering, and cyclostationary algorithms. Cyclostationary processing is particularly appropriate for this architecture because of its computational complexity. This architecture, referred to as the butterfly machine architecture, provides a scaleable solution compatible with workstation environments. A key component of this architecture is the memory design referred to as Split Transaction Memory (STM). STM exploits the specific memory reference stream characteristics associated with cyclostationary processing and provides a throughput to the vector processor that approaches 1.0 for anticipated address patterns. There are two aspects of STM that are of special note.

- STM is an interleaved memory that buffers memory references. The primitive organizational element for buffering within a bank is referred to as a cache element. The use of cache elements within banks provides a more efficient organization than standard buffers when three or more cache elements are called for in each bank.
- STM uses a memory decoding scheme that is optimized for memory reference patterns that are characterized by powers of two. This is

accomplished by using permutation matrices to decode bank numbers. A design methodology is developed for constructing permutation matrices that are designed for address patterns with any constant stride of powers of two that yield near ideal performance for interleaved memory systems. A second methodology is presented that results in permutation matrices that yield near ideal performance for constant geometry radix- $r$  butterfly address patterns. The radix- $r$  butterfly permutation matrices, modified to support constant stride of powers of two address patterns, provide near ideal performance for constant stride and radix- $r$  butterfly address patterns. The third address pattern required for FFT-based vector processing, digit reversal, also yields near ideal performance when the radix of the butterfly is equal to or greater than the number of banks. When this condition is not met, the actual performance varies from near ideal to fair. Theory is developed for steady state throughput and maximum latency for each of the address patterns.

Another unique contribution of this research is an event driven software simulator that provides for analysis of STM memory systems. The STM simulator accepts a description of the STM memory and a memory reference stream. When this memory reference stream is processed, details of each cycle are stored at the register level for later analysis. Post analysis routines provide plots and tables for analysis of the simulation run. Programs have also been developed to generate input address streams for constant stride, radix- $r$  butterfly, digit reversed, and random address streams.



## II. HISTORICAL PERSPECTIVE AND RELATED RESEARCH

### A. THE GENERAL PROBLEM

A well designed computer system is one that exhibits a balance of processing capability and communication bandwidth among the various components, delivered at a favorable cost-performance ratio. This balance is established in the context of an existing technology base. Since the advent of the microprocessor, processor design has been at the forefront of computer architecture. Further, the combination of advances in clock rates made available with improvements in the electronics, and architectural advances such as the issuing of multiple instructions per clock cycle, has resulted in an increasing gap between processor computational capability and the ability for memory systems to provide data at sufficient bandwidth to support these computations for general-purpose processors Comerford [Ref 16]. This chapter will summarize techniques that have been explored to enhance the memory system of computers.

Before proceeding further, it should be noted that the scope of memory design techniques has been strongly influenced by the type of computer architecture under consideration. The advent of a variety of multiprocessor architectures has provided both new challenges as well as opportunities. Classes of computer architectures that will be discussed below in the context of the processor-memory imbalance are the *multiple instruction multiple data* (MIMD) and the *vector processor* architectures.

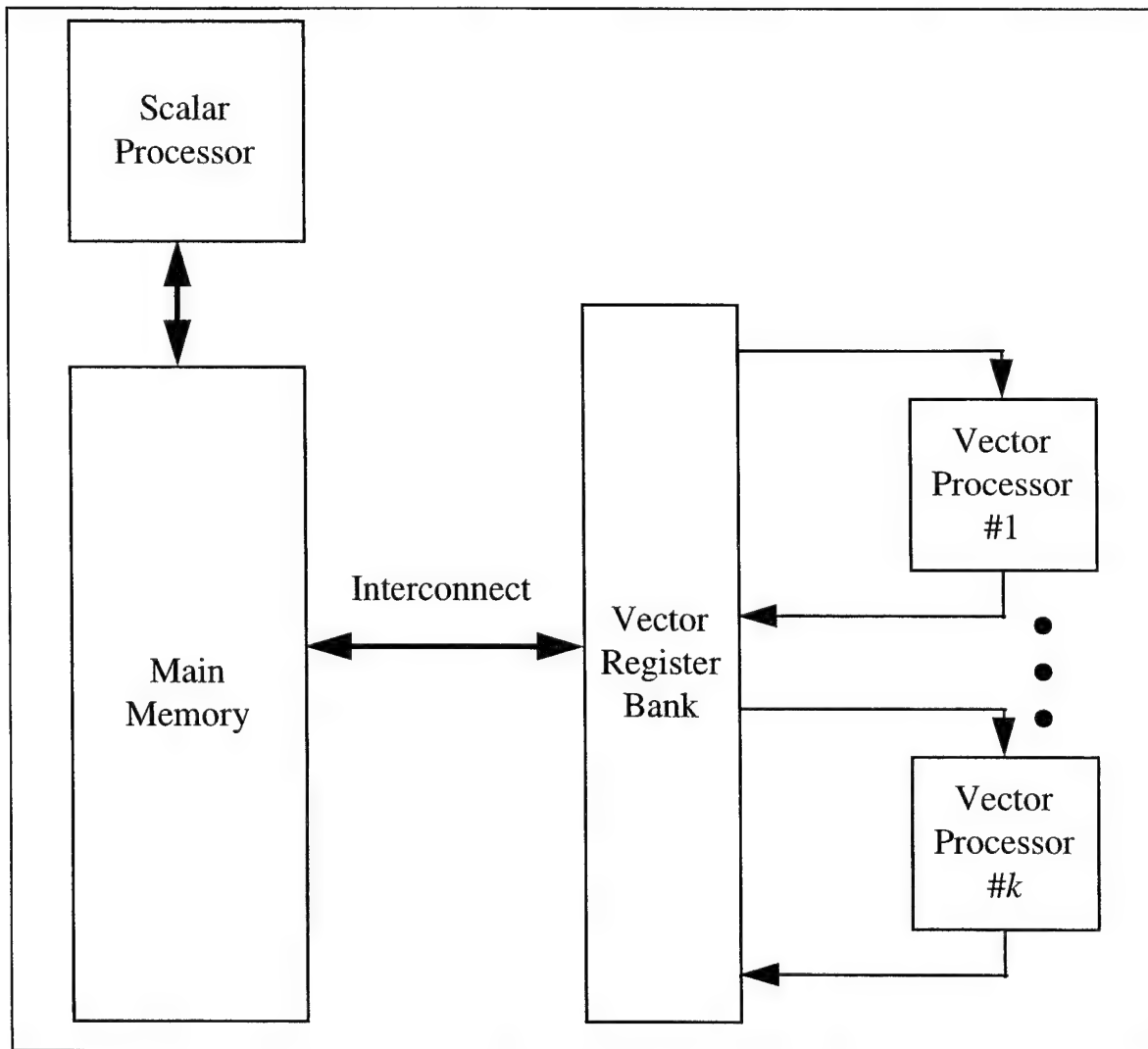
Two prominent MIMD architectures that have evolved and are prominent today are the *distributed-memory architecture*, and the *shared-memory architecture*. The distributed-memory architecture extends the von Neumann architecture by connecting single-instruction single-data (SISD) machines with local and wide-area networks. This provides an alternative to larger monolithic computer systems, namely a system of smaller computers networked together. To the degree that this implies the need for smaller less capable processors in the networked system, the processor-memory imbalance is eased.

The shared-memory architecture is based on two or more processes sharing a memory address space. These processors may be centralized or distributed physically as indicated in Hennessy [Ref 17]. Clearly increasing the ratio of processors to a memory further increases the imbalance between processor capabilities and the corresponding demands on the memory system. However, this architecture provides an opportunity to exploit economies of scale of the memory system. Further, this architecture generates an address stream that is a composite of the address streams generated by the individual processors. This multiprocessor address stream may have characteristics that are exploitable for improving memory performance.

A generic vector processor architecture is shown in Figure II.1. The vector processor architecture usually consists of one or more special purpose vector processors serviced by a memory system. A *vector supercomputer*, such as the Cray Research Y-MP is typically designed with vector processors and also contains one or more general-purpose processors that can operate on scalar values. But as the name suggests, the processor is specially designed to operate upon one or more vectors. Typically, a single processor will accept two vectors and generate a third vector as an output. The resulting output vector of one processor may serve as an input vector to a second processor. This provides for high-level pipelining of the algorithm. Since the operation performed by a vector processor is also typically pipelined, a new piece of data is generally required for each clock cycle. For a vector processor accepting two vectors as inputs and generating a third as output, three memory references are needed each cycle. Further, pipelining the processor allows these processors to operate at higher clock rates than normally found in computer systems. The high clock rate, the need for a data element from each vector each clock cycle, and the existence of multiple vectors provides a substantial load on the memory system.

The preceding discussion suggests that there are many computer architectures and that the features of the particular architecture will have an impact on the memory requirements and design. This dissertation will focus on a variation of the vector processor architecture. This architecture, referred to as the butterfly architecture, will be presented in the next chapter. As a vector processor, it has many of the properties

described for the vector processor architecture above. It differs in that it is exclusively a vector machine (i.e., it does not perform any scalar operations).



**Figure II.1 Generic Vector Processor Architecture**

The remaining portion of this chapter will describe in some detail the two primary techniques used to build memory systems. These techniques are known as cache and interleaved memory systems. Before discussing cache and interleaved memory system, a brief discussion of the characteristics of a *memory address stream* will be presented. Those characteristics that effect the memory address stream will also be addressed.

Much of the material presented in this chapter is a summary of ideas that can be found in many sources. Cache and interleaved memory concepts can be found in Stone [Ref 18]. Another source for cache memory is Hennessy [Ref 19].

## **B. MEMORY ADDRESS STREAM**

In this section, the characteristics of a memory address stream will be described for a general-purpose processor and for a vector processor. Any method used to describe or characterize the memory address stream for the purpose of anticipating future memory references is referred to as a *characteristic*.

The first factor to consider that effects the characteristic of a memory reference stream is the type of processor that is generating the address stream. Two types of processors will be considered here: a general-purpose processor and a vector processor.

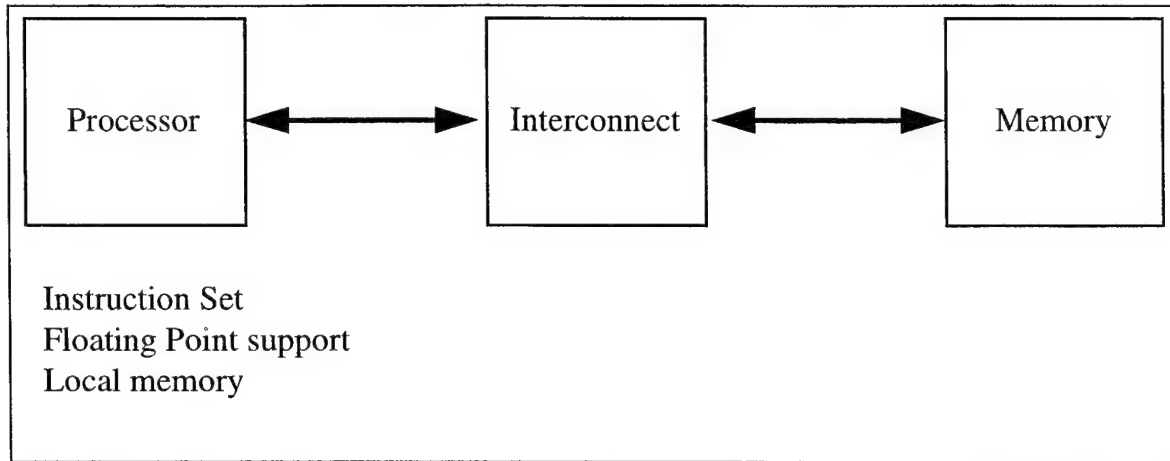
First, the general-purpose processor will be considered. A much utilized example of a memory reference stream characteristic for general-purpose processors is *locality of reference*. It has been postulated and confirmed under many circumstances that addresses close by in the memory space to the most recently accessed memory address, are more likely to be addressed in the near future than those that are not nearby. Another example of a general-purpose processor characteristic is that instruction fetches have a tendency to be sequential or linear (i.e., the execution of a set of instructions that do not contain branches will follow one after the other.)

A model of general-purpose processing architecture (i.e., von Neumann) with the three basic components of processor, interconnect, and memory is shown Figure II.2.

The processor establishes the *de facto* requirements for memory accesses for which the interconnect and memory must respond. Said in another way, the processor is usually thought of as taking the active role generating the memory reference stream. This is accomplished by repeating the execution cycle consisting minimally of a fetch, decode, and execution cycles. The two types of memory references generated by the processor are instruction fetches and data read or write references. As indicated earlier, instruction references demonstrate a linearity property because they are typically segments of



instructions in a program that will execute without branching. Another characteristic of instruction fetches is that they are almost always read only.



**Figure II.2 General-Purpose Processor**

The address stream characteristics are influenced by the following factors:

- characteristics of the processor,
- characteristics of the software development tools, and
- application program characteristics.

Characteristics of the processor include first the *instruction set architecture* (ISA). The majority of the instructions in a *complex set instruction computer* (CISC) architecture such as the Motorola 680X0 series contain memory references as an integral part of the instruction (i.e., one or two memory references are made as a result of the execution of the instruction. This is in contrast to a *reduced instruction set computer* (RISC) ISAs where all memory references are accomplished with dedicated memory reference instructions. The ratio of instruction references to data references is generally higher because the set of RISC instructions is simpler and fewer by design. Therefore, the sequential characteristic is more pronounced.

The number of registers available to the processor also effects the memory address stream. The more registers available, the more variables can be maintained at the processor without read and write accesses back to main memory. For a larger number of registers, the number of memory references will decline in general, and the ratio of the

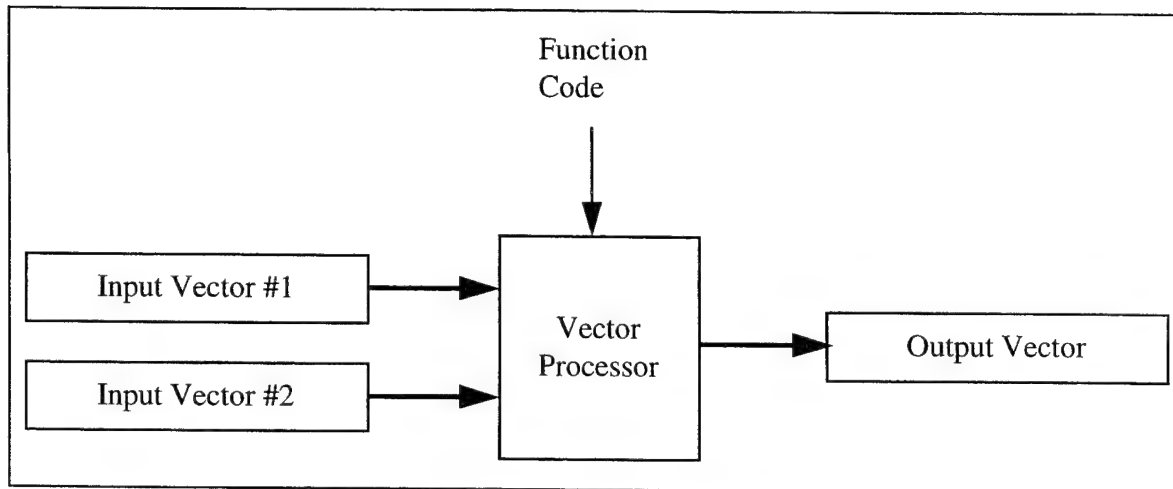
number of instruction references to data references will increase. Most programs however cannot use more than 20 to 30 registers. Thus, the more recent RISC designs are based on 32 registers. When more silicon is allocated to registers such as with the Sun Sparc design, the number of registers for one context is limited to 32 registers. [Ref 20]

Software languages and their corresponding compilers also have an impact on the character of the memory reference stream. Extensive use of looping constructs such as the WHILE statement provides for locality of reference whenever the loop executes multiple times. Also, the longer the loop, the greater the linearity of the memory reference stream. The programming practice of modular decomposition and the use of the function construct also yields locality of reference. Allocation of memory for data also provides some locality of reference. For example, in the C programming language, local variables of a function are stored together. Variable passing using the stack provides some locality of reference. However, dynamic allocation of memory is accomplished from a data structure referred to as a heap. Dynamic allocation can result in variable references to be spread about the address space if they are allocated and deallocated frequently.

The last factor, application program characteristics, provides the biggest uncertainty regarding the memory reference stream. A program language provides substantial flexibility regarding the implementation of a program. Given the particular design decisions of any memory system, it is possible to write an application that will exploit the weaknesses of the memory system.

The other type of processor that will be discussed is the vector processor. A vector processor accepts one or more vectors as input, as well as an operation or function code, that specifies the function to be performed as shown in Figure II.3. The vector processor will accept one data input from each input vector on each clock cycle. Further, the processor will perform an operation repeatedly on a finite number of data points. For example, if the operation was addition, then the processor would add each pair of points from two vectors. A radix-4 operation would perform the butterfly operation on data sets

of four points at a time. In general, input data vectors for a radix- $r$  operation include the data vector and a vector containing twiddle factors.



**Figure II.3 Vector Processor**

An important aspect of the vector processor is that the operation to be performed is by nature repetitive and therefore the need for instruction fetches is sparse relative to data references. Therefore, for practical purposes, the instruction fetches may be ignored in some circumstances.

The data reference stream has several important properties. First, for a given vector operation, a vector is either an input or an output for all data points. Therefore, the memory reference stream will either be a series of reads or writes with respect to the vector. Further, for a given operation, vectors are accessed in a well defined path and not subject to run time decisions. In other words, the memory address pattern for data references for a vector machine are primarily determined at compile time. For one vector operation and the associated data, a vector processor could generate the entire memory reference stream prior to executing the first instruction! This is in sharp contrast to the general-purpose computing case where the next memory reference may be determined by the results of executing the current instruction. At a higher level of program control, there may be conditional branch instructions that may have to be evaluated before a particular vector operation can be executed.

There are three addressing patterns that are of interest for the butterfly machine architecture to be described in Chapter 0:

- *constant stride  $s$ ,*
- *constant geometry radix- $r$  butterfly, and*
- *digit reversal.*

The most common addressing patterns used in vector machines are patterns of constant stride  $s$ , where  $s$  is the spacing between the references. For example, a vector multiply of two vectors would require a constant stride of one for each input vector as well as the output vector.

The constant geometry radix- $r$  butterfly, and the digit-reversed pattern are both used to compute FFTs. The constant geometry radix- $r$  butterfly pattern is composed of a number of constant stride sequences. One pass of the digit reversal pattern is required for each FFT. A discussion of these memory reference patterns can be found in Oppenheim [Ref 21].

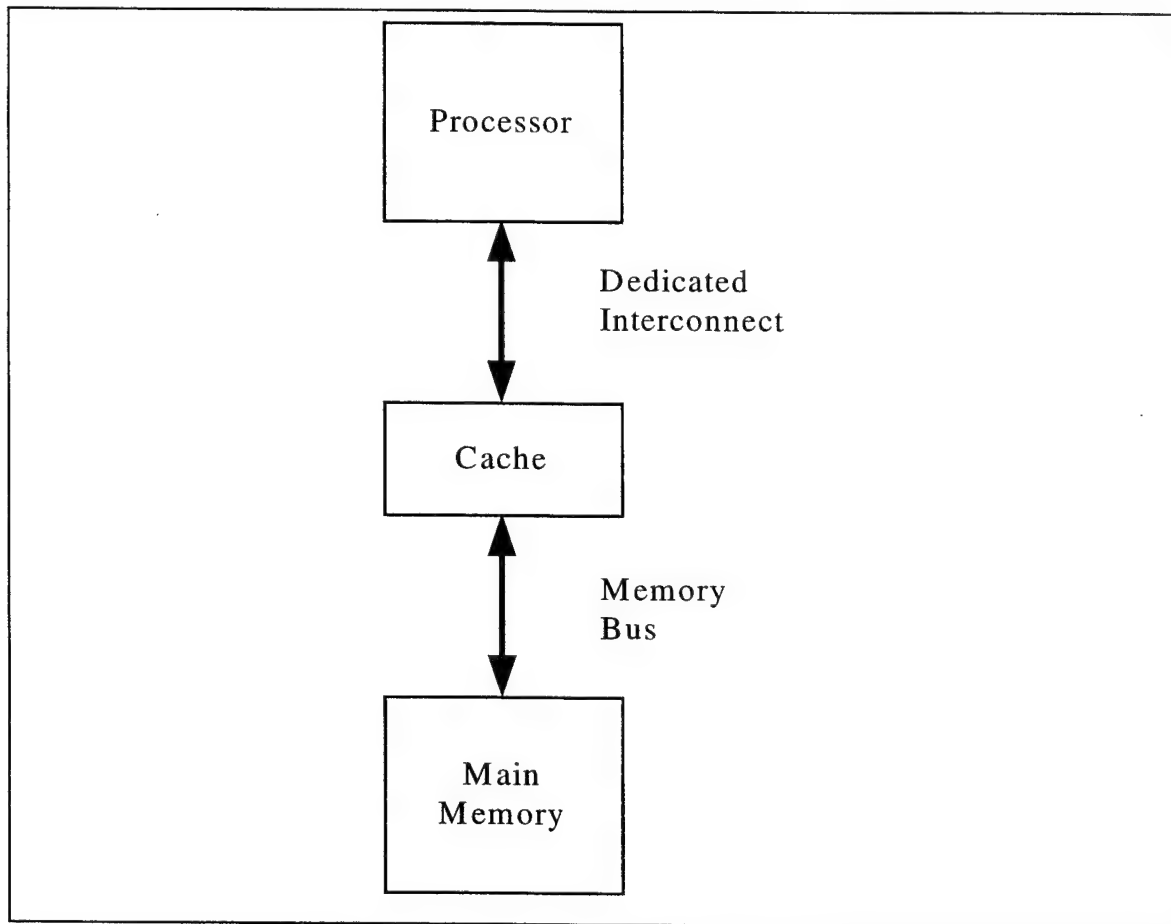
### C. CACHE MEMORY

There are two basic memory enhancement techniques that have been developed to minimize the impact of the processor-memory imbalance, namely cached and interleaved memory. Cache memory is by far the most pervasive because it has been found to be effective when dealing with the general-purpose computer architecture. It is so successful that almost any computer system acquired today will have at least one cache in the memory system and frequently more than one. A cache memory system exploits the locality of reference property described in the section above.

Banked interleaved memory has been used in a general-purpose architecture as a secondary enhancement technique to cache memory. However, it is the primary means for increasing memory bandwidth for vector processor architectures such as supercomputer vector processors.

Figure II.4 illustrates the physical organization of a cache memory system. The cache memory is a small memory when compared with the main memory, but operates at

the same speed as the processor. It logically can be divided into a cache memory, and a cache memory controller. From the processor's interface looking down, the cache looks like main memory where the memory response time is not constant. To main memory, the cache appears to be a bus master that always requests a block of memory references at a time.



**Figure II.4 Cache Memory System**

The cache memory is organized into equal sized blocks referred to as *lines* or *cache lines*. Representative sizes for cache lines,  $l$ , range between 16 and 64 bytes. Main memory is logically organized into blocks of length  $l$ .

Whenever a read-memory reference is made by the processor to the cache, the reference is either contained in the cache or it is not. This is termed a cache *hit* or *miss* respectively. When a program begins, the cache is empty and therefore the first reference is by definition a miss. Under these circumstances, the cache must obtain the memory

reference from main memory. The cache will obtain the entire line associated with the reference, store the line in a cache line, and pass the reference back to the processor. The cache is then ready to process another memory request.

On the next memory reference request, if the request is not located in the cache (i.e., if the request is not located within the cache line that was previously loaded), then the process repeats as before. If however, the reference is contained in the cache, then the cache simply responds with the data. The dedicated interconnect between the cache and the processor will generally allow the data access to proceed at the processor clock rate.

The bandwidth of a cache-based memory system may be modeled in terms of the *effective cycle time*. The effective cycle time  $T_{eff}$  is defined as the average cycle time to access one word when filling a cache line, adjusted for the number of elements of the line not actually used and the number of elements of a line used more than once. This is the effective bandwidth as seen by the processor. The effective bandwidth can be expressed as:

$$T_{eff} = T_{ca} \frac{l}{l_a} \frac{l}{l + l_r}, \quad (\text{II.1})$$

where,

$$T_{ca} = \frac{T_c}{l}, \quad (\text{II.2})$$

and

$T_c$  is the time required to fill a cache line,

$l$  is the number of words in a cache line,

$l_a$  is the number of words in a cache line that are accessed by the processor, and

$l_r$  is the number of times that words in a cache line are accessed by the processor after the first access (i.e., repeat accesses).

The first term of Equation (II.1),  $T_{ca}$  represents the average cycle time to access one word. The second and third terms reflect adjustments based on the degree that the locality of reference property is present. The second term is the fraction of the words in

the line actually utilized by the processor. If all of the words are accessed, then this reduces to unity. If a fraction of the words are used, then the average cycle time is adjusted by the reciprocal of that fraction. The third term reflects the benefit for reusing a word without having to fetch it back from main memory. If no words are reused, the term reduces to unity. If each term was reused once, then the term would be one half. Equation (II.1) illustrates that the effective bandwidth provided by the cache can vary substantially in either direction from the average cycle time to load a cache line, depending upon the locality of reference.

Caches are classified based on the kind of information that is to be cached. There are instruction caches, data caches, and combined caches. An instruction cache is easier to build because the instruction fetches are read-only and therefore the hardware necessary for maintaining consistency between the instructions in the cache, and instructions in main memory is not necessary. Further, if it is determined that the locality of reference is different for instructions and data, then separate data and instruction caches can be better tailored to their respective needs. However, a combined data and instruction cache can use the cache resources efficiently.

A short discussion on the time-varying characteristics of cache is in order before leaving the discussion of cache memory. When a process begins, none of the process's instructions or data is contained in the cache. Most of the initial references are misses, but as the process progresses, more and more of the program and data necessary for the process are loaded into the cache. The contents of the cache are "demand driven" by the processor's references. A point in time is reached where almost all of the address references are in the cache and therefore most references are cache hits. This assumes that the cache has sufficient capacity to support the process. The period of time between the start of the process, until the process is mostly cache hits is referred to as the *transient* time. That period of time beginning with mostly cache hits is referred to as *steady-state* time. When a process transitions to another part of the program, or when another process's context is switched in, then another transient is experienced followed by a steady-state period.

A process's memory address space as well as the cache lines can be illustrated graphically. The area of the memory address space that is contained in the cache or alternatively, those cache lines that contain the process's references at the point of steady state, is referred to as the process's *foot print*. If such a graphic were available and updated in real time, it would show at any instant that portion of the memory space that is active. The time-varying dynamics of the memory address space and the cache would be viewed through continually updated graphics throughout the life cycle of the processes. [Ref 22]

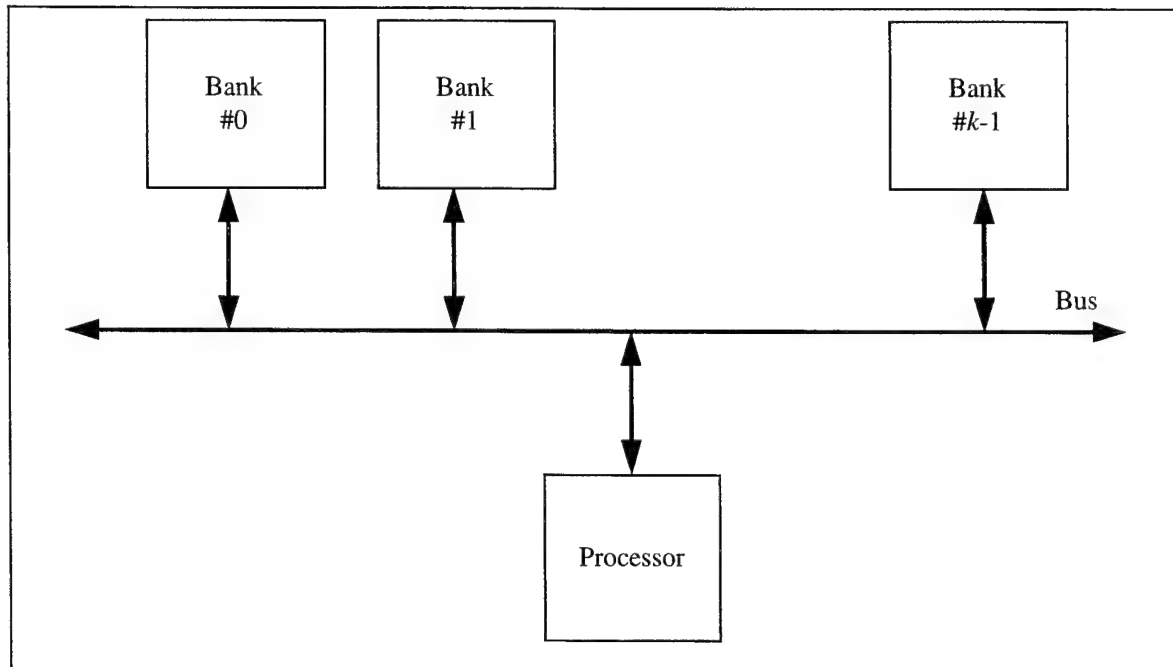
#### **D. INTERLEAVED MEMORY**

The other memory management technique to be described is interleaved memory. A block diagram of a banked interleaved memory is shown in Figure II.5. Memory devices (e.g., DRAMs) are mapped into the address space such that the memory address space is partitioned evenly among the banks. The primary parameters that define an interleaved memory scheme include the *number of banks* and the scheme for mapping memory addresses to a bank number and index within a bank pair. This will be referred to as the *bank number decoding* and *bank index decoding* schemes respectively. In general it is desirable to have a large number of banks since the potential data rate is greater. Electrical properties such as fanout suggest a cost associated with more banks and therefore a cost benefit tradeoff must be evaluated for a given application. As is indicated in the discussion below, the bank number selection criteria also may have an impact on the number of banks chosen.

The following is a brief description of the operation of a banked memory system that incorporates interleaving to increase memory performance. A bank will accept a memory request if the bank is not processing a previous memory request. Therefore, as many as  $k$  memory requests can be pending at a time (i.e., one from each bank). If a busy bank is selected (i.e., if it is processing a previous memory request) then the memory system *stalls*. A memory system is said to stall when the current memory request is not accepted. No other memory requests will be allowed until the selected bank has



completed the current memory request. The stalled memory request is then accepted and the process continues.



**Figure II.5 Interleaved Memory Block Diagram**

If each of the banks can be kept busy, then a total memory bandwidth of  $kB$  words per second can be obtained from the memory system where  $k$  is the number of banks and  $B$  is the bandwidth of a single bank. In order for this to occur, all banks must be continuously processing memory access requests and the memory ratio must be less than or equal to the number of banks. This is accomplished for example, if the banks are selected in a round robin fashion (e.g., 0, 1, 2, 3 ...  $k-2, k-1$ , 0, 1, 2, ...  $k-2, k-1$ , 0, 1 ... where  $k$  is the number of banks). The effectiveness of interleaved memory is then directly related to the ability to keep the banks busy which is accomplished by providing a work distribution that is approximately uniform over time.

The three primary performance measurements of interest for interleaved memory systems in this effort are:

- *latency* ( $L$ ),
- *throughput* ( $TP$ ), and

- *speedup* ( $S$ ).

Latency ( $L$ ) is defined as the number of memory cycles from the time a processor attempts to issue a memory reference request, until the time the request is completed. Note that latency contains two basic components. First, latency occurs due to the delay in the memory bank necessary to service a memory request. Second, latency will increase if the memory system is saturated and therefore the memory system does not accept additional memory references.

Latency is also time-varying, and can be measured at each point a memory response is completed. This time-varying view of latency can also be depicted graphically. Scalar measures of latency include maximum latency ( $L_{\max}$ ), average latency ( $L_{\text{avg}}$ ), and the standard deviation of the latency ( $L_{\text{std}}$ ).

The memory ratio ( $MR$ ), introduced in the cache memory section, is directly related to latency in interleaved memory systems. The minimum latency for an interleaved memory system is the memory ratio plus any overhead related to the interleaved memory system. Interleaved memory systems generally use registers to receive an input and for placing data onto the bus for read requests. This adds two cycles to the minimal latency and therefore the minimum latency for an interleaved memory system will be

$$L_{\min} = MR + 2. \quad (\text{II.3})$$

The throughput is defined to be the ratio of the total number of memory cycles required for an ideal memory device to complete a set of memory references, to the actual number of memory cycles used to complete the set of memory references for a particular memory design. An ideal memory device is defined as one that can service a memory reference in one cycle. Throughput may be expressed as:

$$TP = \frac{C_{\text{ideal}}}{C_{\text{actual}}}, \quad (\text{II.4})$$

where:

$C_{ideal}$  is the total number of memory cycles for a given task, for an ideal memory device. This is equivalent to total number of memory references.

$C_{actual}$  is the actual number of memory cycles necessary to complete the same task and,

The memory design can never be better than the ideal memory device, therefore

$$0 \leq TP \leq 1. \quad (\text{II.5})$$

Throughput is a measure of how well the processor is serviced by the memory system. In this analysis, it is assumed unless otherwise stated, that the processor will issue one memory access per system clock cycle unless the memory system blocks the request. Therefore, a throughput of 1.0 indicates that the processor is provided one “memory response” for each clock cycle.

Another measure of throughput is the *steady-state throughput*. In a manner analogous to cache memories, there is a period of time when the memory goes through a transitory period which is reflected in an irregular output. This is followed by a period where output is periodic (e.g., frequently a constant). These two periods will be referred to as the transient and the steady-state response of the interleaved memory system. Of particular interest are:

- The *length of the transient response* ( $T_{tr}$ ). It is desirable that this figure approach the minimum latency, and that it be a small fraction of the length of the vector processed.
- Steady-state throughput ( $TP_{ss}$ ). The steady-state throughput is a better measure of throughput because it eliminates the effects of the transient. However, this measurement is only valid when the transient response is a small fraction of the vector length as indicated above.

Speedup is a performance measure that focuses on the relative improvement gained when adding additional memory components. Speedup ( $S$ ) is defined as the ratio

of number of memory cycles necessary to complete a given task using one memory bank to the number of memory cycles necessary to complete the same task using  $k$  banks or

$$S = \frac{C_1}{C_k}, \quad (\text{II.6})$$

where:

$C_1$  is the number of memory cycles required for one bank, and

$C_k$  is the number of memory cycles required for  $k$  banks.

Note that  $C_1$  is the product of the number of memory references,  $C_{ideal}$ , and the memory ratio ( $MR$ ). It can be seen that the relationship between throughput and speedup is a mutiplicative factor of the memory ratio as shown below:

$$S_k = \frac{C_{ideal} \cdot MR}{C_k} = TP \cdot MR. \quad (\text{II.7})$$

Both throughput and speedup are performance measures of memory bandwidth. Both will be viewed as a scalar measure of performance as defined by the formulas above. Throughput may also be defined as a moving average, capturing the time-varying quantity of throughput. This can be illustrated as a line graph.

One characteristic of bulk memory that has motivated the use of banked interleaved memory is the difference between the memory access time ( $t_a$ ) and the cycle time ( $t_c$ ). Many devices (e.g., DRAMs) have a cycle time that is greater than the access time because of overhead tasks that must be completed prior to beginning another access. For example, a read operation to a DRAM memory cell destroys the contents of the cell. The original contents must be written back to the cell to preserve the value. The relationship between the access and cycle times can be expressed as

$$t_c \leq k \cdot t_a \quad k = 2, 3, 4... \quad (\text{II.8})$$

If the number of banks is selected such that

$$B \geq k \quad (\text{II.9})$$

where  $B$  is the number of banks, then the overhead time can be absorbed if all banks can be kept busy. The memory can then operate at the access rate rather than the cycle rate.

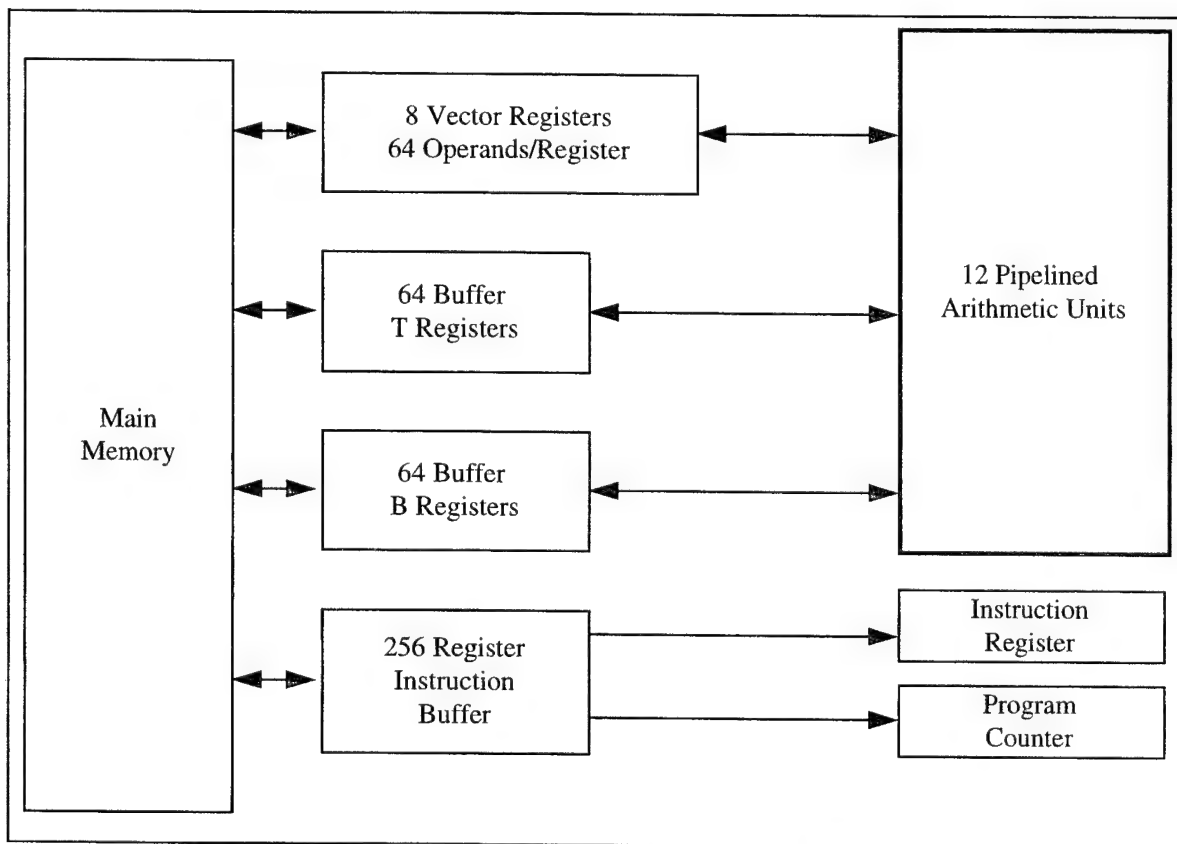
Although banked interleaved memory has been used to enhance a cached memory scheme, the number of banks is small typically two or four. Generally, an interleaved memory with a large number of banks is used for vector processing. One category of vector processing is supercomputer vector machines such as the Cray I, the Burroughs Scientific Processor (BSP), and the Convex C3800. Another category is the attached vector processor. An example of an attached vector processor is Floating-Point System's FPS-164 [Ref 23].

Two important characteristics of vector processors include the ability to perform scalar operations and a memory system that is hierarchical. The need for scalar operations is clear in a supercomputer where a relatively large computational problem, is expected to be solved without additional computer support.

The Cray I, illustrated in Figure II.6, is an example of a supercomputer with a memory hierarchy, and the ability to perform scalar as well as vector operations. There is one main memory which provides for the majority of storage. The fastest memories are connected directly to the pipelined arithmetic units. The Arithmetic Units perform scalar as well as vector operations. By placing the highest speed memories next to the processor, the processor can operate at an optimal speed so long as the data is contained in the high-speed memories.

This is valid when the algorithm can be written in such a way that data is repeatedly accessed before returning to main memory. Alternatively, it can be said that the data has locality of reference at a high level of granularity. However, in this instance, the programmer is responsible for managing all of the levels of memory (i.e., it is not accomplished automatically as was done with cached memory in the general-purpose computing case).

When an attached processor is used in conjunction with a personal computer or workstation, it is reasonable to expect that the workload can be partitioned such that a part of the algorithm will be executed on the host's general-purpose processor, the remainder on the vector processor. In order to reduce complexity and cost, the vector machine may be optimized to perform vector operations and therefore mitigate the need for scalar operations on the vector processor. The lack of scalar operations will in turn, reduce the likelihood of repeated use of data before returning it to the main memory. This in turn suggests that memory schemes without a hierarchy may be appropriate for an attached vector processor.



**Figure II.6 Cray I Memory Hierarchy [Ref 24]**

Interleaved memory systems are also designed to take advantage of a characteristic of the memory reference stream. Therefore, as was the case for cache memory systems, the performance of the interleaved memory system is highly dependent on the particular program executed.

For example, it has been observed that a purely random addressing pattern has a speedup that can be expressed as

$$S = \sum_{k=1}^B \frac{k^2(B-1)!}{B^k(B-k)!} \approx B^{0.56} \quad (\text{II.10})$$

where  $B$  is the number of banks [Ref 25]. This is a disappointing result, given that the bandwidth is proportional to the square root of the number of banks. This result, coupled with the large values for latency, has discouraged the use of interleaved memories with a large number of banks in general-purpose computing.

However, the memory reference pattern based on accesses to vectors is quite different than a memory reference pattern generated by a general-purpose computer. These patterns are *deterministic* and they are characterized as having patterns with constant stride. Operations such as vector addition and multiplication have a constant stride of one. Other operations have constant strides other than one. More complex address patterns are found with operations such as a radix- $r$  butterfly and digit reversal. However, these more complex patterns have multiple series of constant stride embedded in the address pattern. A model for memory address patterns, as they related memory performance, is presented in Chapter V.

Several *memory decoding* schemes will be described below. Memory decoding for banked interleaved memory systems includes determination of the bank number and the index within a bank. Frequently, the index within a bank is accomplished in a straight forward manner using a subset of the address bits. The primary focus of the discussion below will be in the selection of a bank number. The motivation for the different bank selection schemes is to find a scheme that will spread memory references evenly to all of the banks (i.e., in a round robin pattern), for the memory address patterns most likely to occur. It is also desired that the bank selection scheme have the following properties:

- an implementation that is inexpensive in terms of hardware
- have a small propagation delay, and

- imposes the fewest restrictions on the number of banks.

The simplest decoding or interleaving scheme of the memory space uses the least significant bits to directly select a bank, and the remaining higher order bits to select a word from the selected bank. **This will be referred to as the *conventional decoding or interleaving scheme*.** Conventional decoding has no implementation requirements but requires the number of banks to be a power of two. From a performance perspective, an address pattern with a constant stride of one will result in a round robin selection of the banks for optimal utilization of the banks when the banks are decoded using the conventional scheme. However, only a subset of the banks will be selected whenever the stride is not relatively prime to the number of banks. Specifically, for a given stride  $s$ , the number of banks that will be selected is:

$$B_{eff} = \frac{B}{\gcd(B, s)} \quad (\text{II.11})$$

where

$B$  is the total number of banks in the memory system,

$s$  is the stride of a constant-stride address pattern,

$B_{eff}$  is the *effective number of banks*. By effective, it is meant that an effective bank is one that is actually given memory references for the specified address pattern.

$\gcd(a,b)$  is the greatest common divisor for  $a$  and  $b$ .

A bank that is referenced for a given addressing pattern is referred to as an *effective bank*. For example if the stride equals the number of banks (or a multiple of the number of banks) a single bank will receive all of the memory requests regardless of the number of banks in the memory system. The effect on the number of banks, stride, and bank selection criteria will be described in detail in Chapter V. Given the problems noted above with strides that are not relatively prime to the number of banks, coupled with the fact that many algorithms such as the fast Fourier transform frequently use powers of two strides, other bank selection criteria have been investigated.



*Linear data skewing* schemes have been proposed where the memory bank selection for a data element contained in an array at location row column indices  $i,j$  is mapped to bank  $ip_1 + jp_2$ . This method is hampered by the need for arithmetic operations to compute the bank number. The hardware is relatively more complex, but more importantly the time needed to compute the bank number has a negative impact on memory performance. However, one data skewing scheme referred to as 1-Skew, has an implementation that requires only logic operators. For an address  $i$ , the bank number is computed as:

$$B_i = \left( i + \left\lfloor \frac{i}{B} \right\rfloor \right) \text{mod} B, \quad (\text{II.12})$$

where

$B_i$  is the computed bank number,

$i$  is the memory address,

mod is the modulus operator, and

$B$  is the number of banks.

Note that the division and the modulo operations are trivial when  $B$  is a power of two. This leaves only the addition operation to perform.

Considering Equation (II.11), it can be seen that if the number of banks in a system is a prime number, then the number of effective banks would always be equal to the number of banks except when the stride is equal to a multiple of the number of banks. The biggest problem with using a prime number of banks is that a direct implementation of such a scheme is requires arithmetic operations that are expensive and incur more propagation delay than is tolerable for performance. Several techniques have been proposed to mitigate this problem. In general the following equations are used to compute the bank number  $B_i$  and index into the bank  $I$ :

$$B_i = i \text{ mod } B, \quad (\text{II.13})$$

$$I = \left\lfloor \frac{i}{B} \right\rfloor \quad (\text{II.14})$$

where

$i$  is the address, and

$B$  is the number of banks.

The Burroughs Scientific Processor used a scheme that reduced the complexity of bank selection to a single adder plus logic operators. This approach to memory selection logic is accomplished in part by selecting a smaller value of  $B$  in Equation (II.14) than in Equation (II.13). For a smaller value of  $B=B'$ , this results in the loss of

$$\frac{B - B'}{B}$$

of the memory. [Ref 26]

An alternative method pipelines the computation of the bank selection [Ref 27]. This approach is dependent on a constant-stride address pattern. The proposed architecture described in Chapter 0 requires addressing patterns that are not strictly constant stride.

The last bank selection technique to be reviewed is *permutation-based* interleaving [Ref 28]. Permutation-based interleaving is based on the same principles as Hamming error detection and correction codes. The bank number  $M$  is calculated using the matrix equation:

$$\mathbf{b} = \mathbf{P} \cdot \mathbf{a}, \quad (\text{II.15})$$

where

$\mathbf{b}$  is a  $(k \times 1)$  column vector representing the bank number,

$\mathbf{a}$  is a  $(r \times 1)$  column vector representing some number (possibly all) of the bits of the memory address, and

$\mathbf{P}$  is a  $(k \times r)$  matrix that specifies the bank number mapping.

The matrix multiplication indicated in the equation is similar to matrix multiplication except that the multiplication operations are logical ANDs and the summation of product terms is a logical exclusive OR. Note that this scheme requires

only logical operations and therefore can be implemented with little propagation delay. Further, a wide array of mappings can be specified. Note that the conventional bank decoding scheme is a subset of permutation-based scheme when the  $\mathbf{P}$  matrix is an identity matrix or rank equal to the number of bits in the bank number. Bank selection using permutation-based techniques are explored further in Chapter 0 Section E.

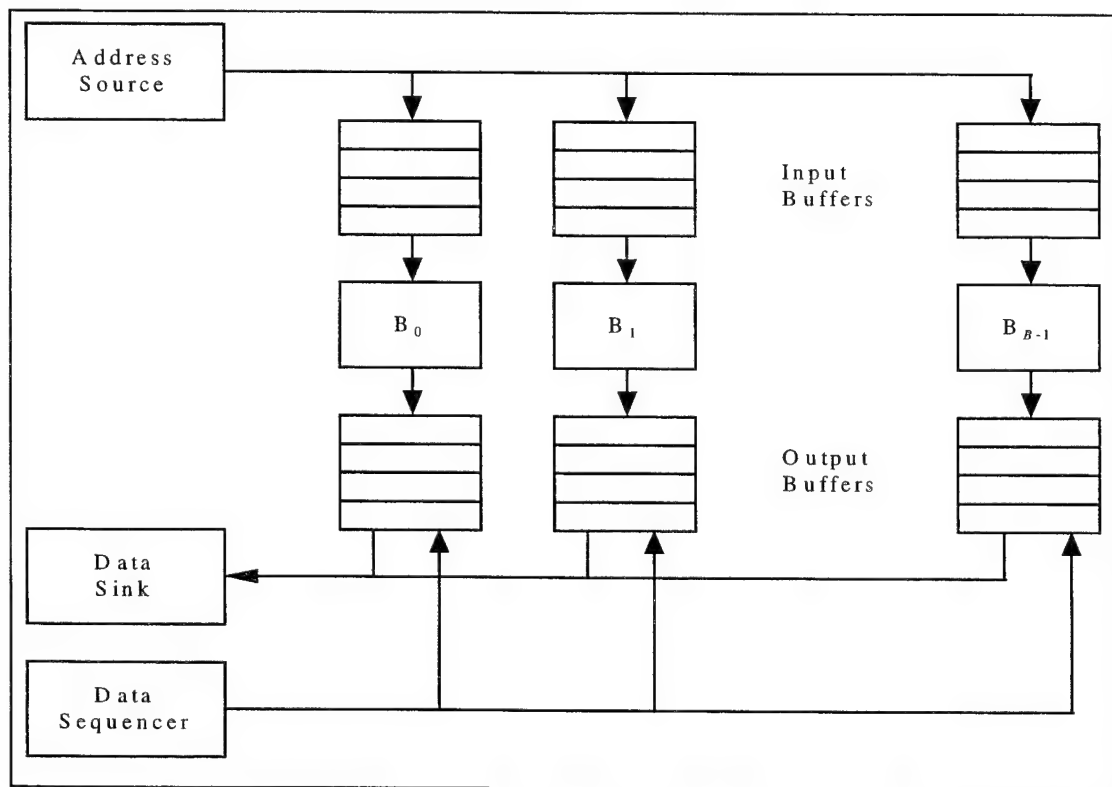
It has been shown that for constant-stride address patterns where the stride ranges from two through 64, Skew-1 and Permutation-Based bank selection schemes have substantially better performance than conventional bank decoding. Permutation-Based bank selection has slightly better performance than Skew-1. The performance measurement in this study was throughput. [Ref 29]

An enhancement to interleaved memory architecture is the use of input and output *buffers* for each memory bank. An interleaved memory model with input and output buffers is illustrated in Figure II.7. An input buffer of length  $b_{in}$  is a first-in first-out (FIFO) queue that allow a memory bank to accept  $b_{in}$  memory requests (i.e., a memory bank can accept memory requests without completing a memory request that is currently in progress. **A standard interleaved memory architecture is defined in this work to be one that has one input and output buffer.** A memory system with more than one butter is referred to as a STM memory.

Buffers are useful for smoothing out irregularities in the memory address pattern. They do not improve throughput if there is an insufficient number of effective banks. To illustrate, consider a standard interleaved memory system with eight banks and an effective memory ratio of eight. For a stride of two, one possible bank selection pattern is

$$\{0,2,4,6,0,2,4,6,\dots\}.$$

There are four effective banks and the resulting throughput will be 0.5. Adding buffers will not improve throughput regardless of the number of buffers added since the four banks that are in use are effectively used 100 percent of the time.



**Figure II.7 Interleaved Memory With Queues [Ref 30]**

Compare this to the situation where the same interleaved memory system is presented an address pattern such that the bank selection pattern is

{0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,0,0,1,1,...}.

The first memory reference will be accepted followed by a stall because bank 0 is busy with the first request. Once the first request is completed, the second memory reference is accepted by bank 0. On the next cycle, the third memory reference is accepted followed by a stall. This pattern continues until the vector is processed.

Now consider the case where each memory bank can accept two memory requests (i.e., the bank can be processing one memory request, and accept another. In this case, each memory bank will accept the two memory requests on the first cycle. Since 16 cycles will have passed between the time bank 0 received the first memory reference, and the time that the second cycle begins, each bank will have sufficient time to process the memory requests as they are presented. The throughput will be optimal in the steady state. Observe that this use of buffers will increase latency.

The Split Transaction Memory (STM), described in Chapter IV, incorporates the concept of buffers in interleaved memory. A high-level view of STM is shown in Figure II.8. Each memory bank consists of three components:

- The bulk storage module is the device that provides for data storage. The bulk storage module contains one or more chips (DRAMs with current technology) and the refresh circuitry.
- The cache elements are high-speed memory that serves as an intermediate staging area for data requests from the processor, and memory responses from the bulk storage module.
- Controllers for the interfaces between the bulk storage module and the cache elements, and the interface between the cache elements and the memory bus.

On each cycle, a STM module may perform none or all of the following operations:

- Accept a memory request from the processor. A memory request is accepted if the bank is selected and if the bank is not full (i.e., there is room in the cache element for another request). Accepting a read or a write request requires that the address or address and data be stored in a cache element respectively.
- Manage the bulk storage module. This includes issuing memory requests to the bulk storage module and accepting data from a memory read request.
- Placing data on the bus in response to a memory read.

The cache elements are managed as a circular queue with three indices. The first is used for marking the next free cache element available for accepting new memory requests. The second index is used to track which memory request should be processed by the bulk storage module. The last index points to data associated with a processed memory read output.

The key difference in buffers and cache elements is the organization and use of registers. As illustrated in Figure II.9, both buffers and cache elements are used to facilitate the transfer of data between the bulk store in a bank and the bus. However, a buffer pair uses one data register for the input buffer and another data register for the output register. A single buffer provides pipelining of memory requests since a new memory request can be placed in the input register in parallel with the bulk store providing a memory response in the output register. On the other hand, a cache element only has one data register and therefore two cache elements would be required to provide pipelining as described above for a single buffer.

A comparison of the representative storage requirements for buffers and cache elements is also shown in Figure II.9. Both schemes must store the address and provide administrative data to maintain the sequential ordering of the memory requests. Two indices are needed when maintaining an input and output buffer scheme as shown in

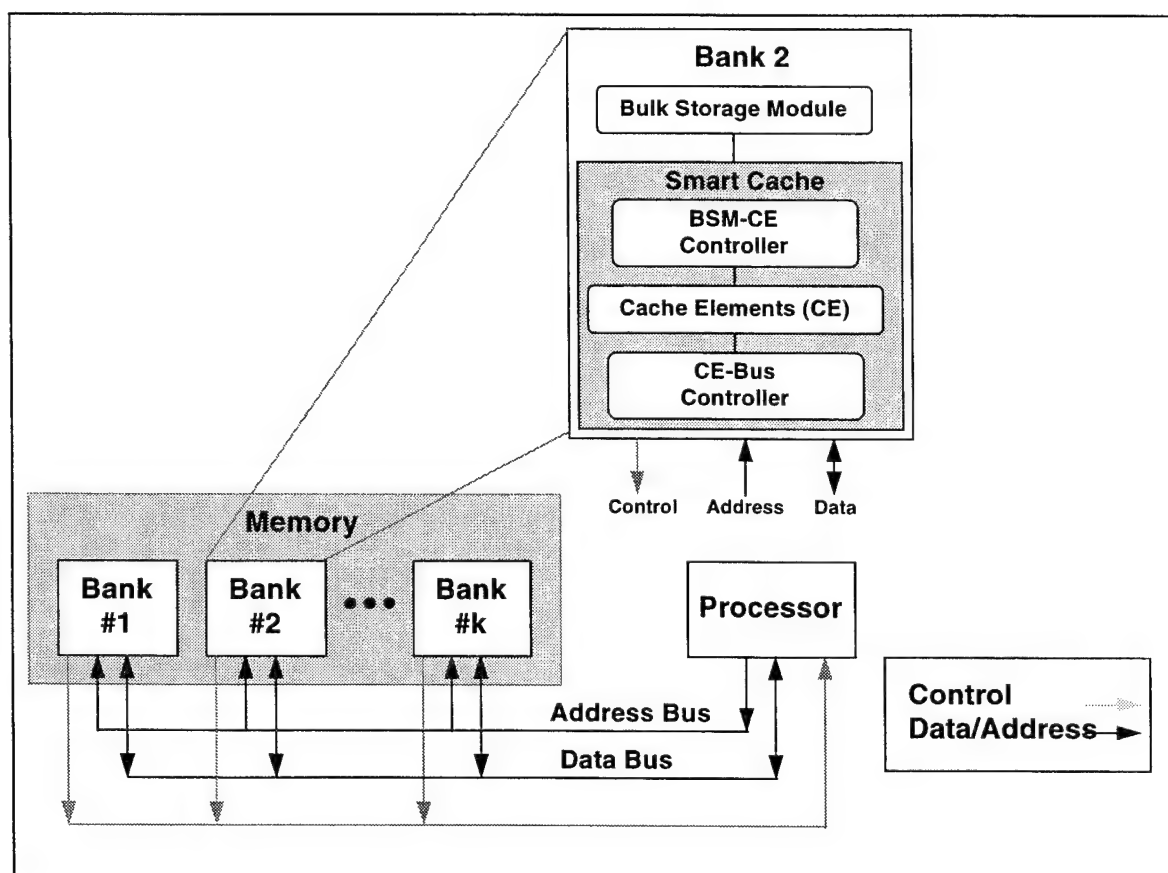


Figure II.8 Split Transaction Memory Overview

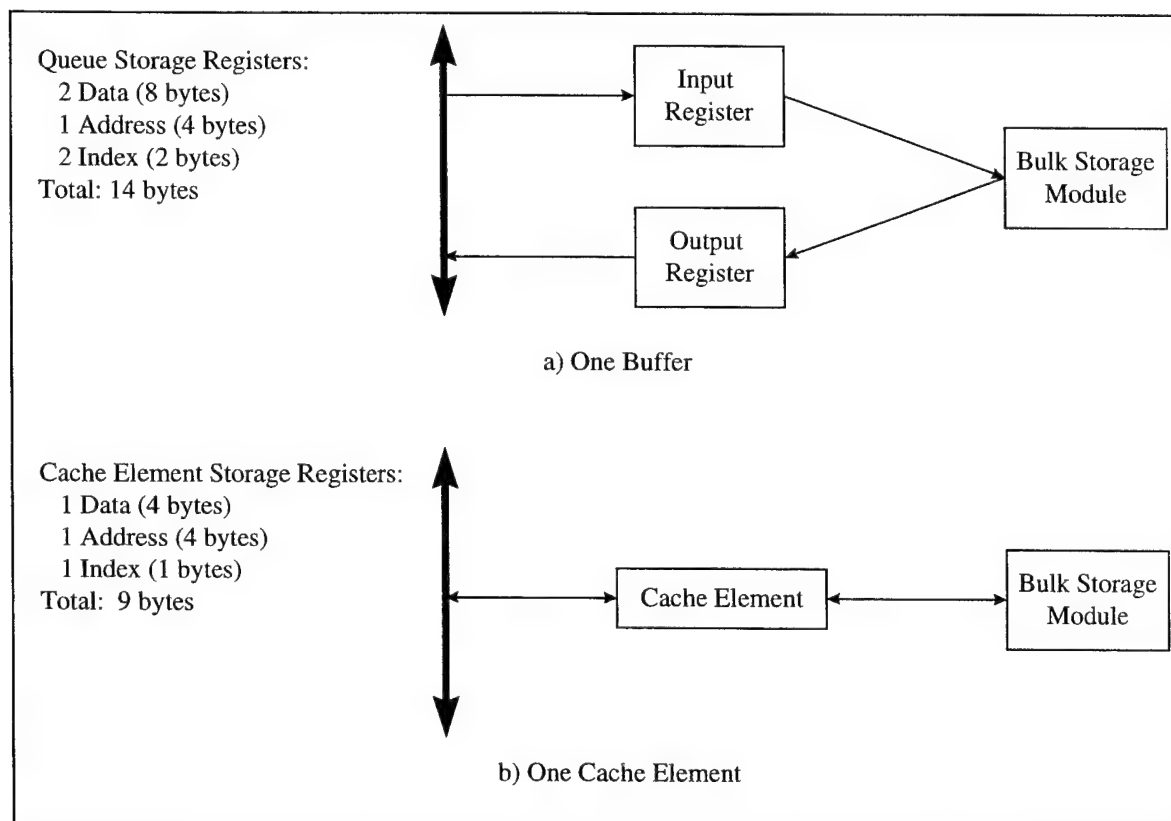
Figure II.7. For purposes of comparison, it is assumed that addresses and data are stored as four byte words and the indices and other control data is contained in one byte.

Implementation of standard interleaving (i.e. using a single buffer) is more efficient using the buffer organization since one buffer requires 14 bytes. To obtain the equivalent pipelining with cache elements requires two cache elements and therefore 18 bytes of storage. However, if a design calls for  $k$  levels of buffering, the cache element scheme becomes more efficient for even small values of  $k$ . In general,  $k+1$  cache elements are required to obtain the equivalent level of pipelining with  $k$  buffers. For example for  $k=2$ , 28 bytes are needed for the buffer scheme versus 27 bytes for the cache element scheme. The number of cache elements needed for a memory system is explored in detail in Chapters V and VI.

Extensive research has been conducted in the area of interleaved memories. One focus of this research has been the nature of the address stream. Early work includes Hellerman [Ref 31] which is based on a random address stream. Later efforts include Chang [Ref 32] and Rau [Ref 33] which provide several dependency models of the data. Several studies have proposed architectural enhancements such as the separation of instruction and data accesses to the memory system Coffman [Ref 34]. Burnett [Ref 35] and Dbois [Ref 36], and Sohi [Ref 37] have investigated different uses of buffers. Multiprocessor structures are analyzed in Baskett [Ref 38] and Briggs [Ref 39]. Fault tolerance is described in the context of interleaved memories in Cheung [Ref 40].

In the next chapter, an architecture for an attached vector processor designed to compute spectral correlation functions will be described. The need for an efficient low-cost memory system will become clear as this design is described.





**Figure II.9 Comparison of Buffers Versus Cache Elements**



### III. BUTTERFLY MACHINE ARCHITECTURE

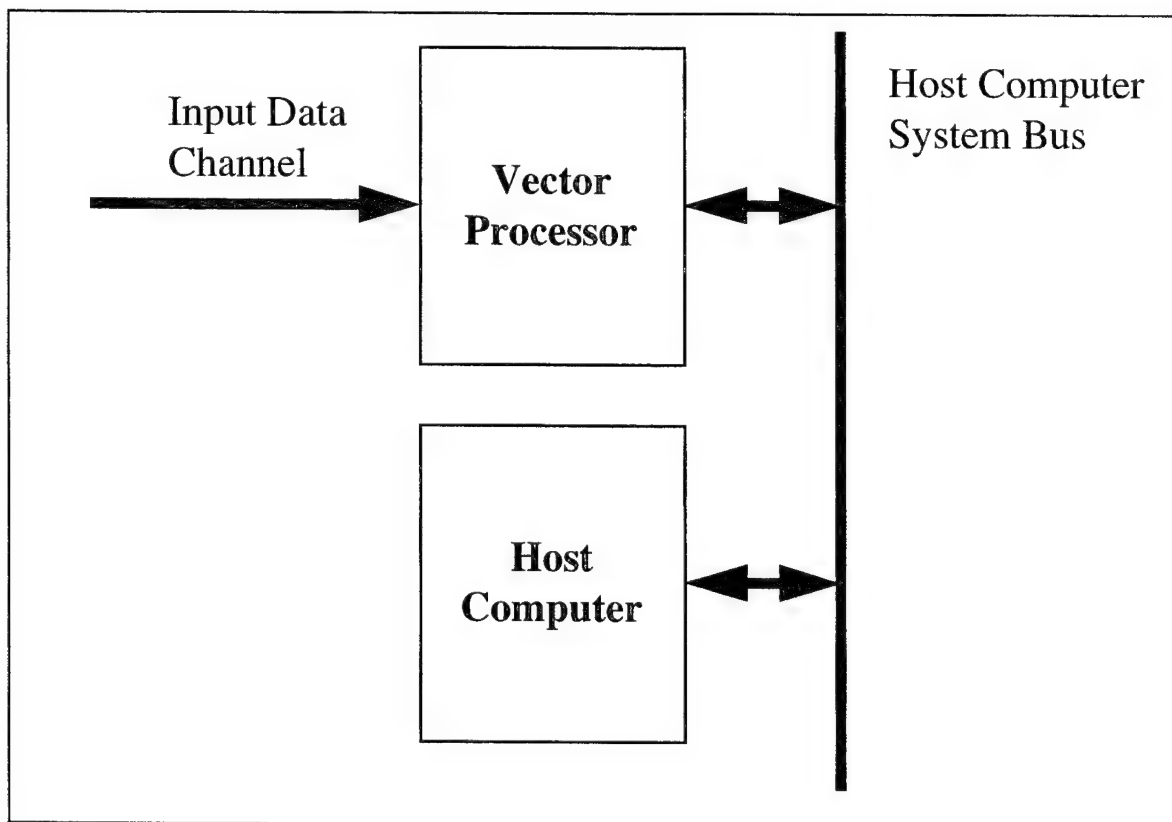
#### A. INTRODUCTION

The following is a description of the butterfly machine architecture. Much of the material is summarized from previous work in Loomis [Ref 41] and Bernstein [Ref 42]. The butterfly machine architecture was developed to provide a high-performance, low-cost solution for cyclostationary processing in particular, and other digital signal processing algorithms that lend themselves to vector operations in general. The butterfly machine is designed to perform only vector processing (i.e., no scalar operations). To obtain high performance, the objective is to approach vector computations with no stalls in the pipeline.

In the following discussion, the term radix- $r$  is used as a parameter of the fast Fourier transform (FFT) algorithm as described by Oppenheim in [Ref 43] and not to be confused with the floating point representation of the hardware. The value of  $r$  indicates the number of inputs and outputs generated with a single butterfly operation. The floating point representation is not discussed but assumed to be 32 bit IEEE-754 format.

VLSI technology has made it possible to develop specialized digital signal processing (DSP) chips that perform FFT butterfly operations for a variety of radices in real time with some latency. When relatively high radices are used compared to radix-2, FFTs can be computed at substantially faster rates than are possible with traditional processors. These processors are also well suited for performing vector operations on data. A computer architecture composed of such DSP chips can compute the vast majority of operations required for cyclostationary algorithms. An architecture is proposed that takes advantage of these specialized DSP chips (referred to as butterfly machines (BFMs)). An architecture using one BFM is defined and is referred to as the one-chip architecture. An implementation of the cyclostationary algorithm, Strip Spectral Correlation Algorithm(SSCA) is illustrated using the one-chip architecture. The one-chip architecture is then expanded into a parallel architecture. Examples of this type of chip technology can be found in the literature [Ref 44], [Ref 45], [Ref 46], [Ref 47].

A block diagram of the processing environment for the butterfly machine is shown in Figure III.1. The host computer is responsible for basic process coordination and scalar operations. The butterfly machine can operate in two modes. In the first mode, the butterfly machine waits for requests from the host computer. When a request is received from the host computer, the butterfly machine responds by accepting data, processing the data, and then sending the processed data back to the host. The input data can come from either an external data channel referred to in Figure III.1 as the input data channel, or from the host computer via the system bus.



**Figure III.1 Butterfly Machine Environment**

In the second mode, the butterfly machine performs a function on a stream of data, sent to the butterfly machine in data sets. For example, the data could originate from sampled data from the input data channel. The resulting processed data is then sent to the host for display and analysis. The butterfly machine program is provided by the host to the butterfly machine via the host computer system bus. What constitutes a program for the butterfly machine will be described below.

There is a traditional tradeoff between specialization of hardware and the scope of functions that can be performed by the architecture. This architecture represents a continuation of a trend toward specialization of silicon to a problem domain. Presently, there are several chips that have been tailored to DSP applications. Notable examples include Intel's i860, Texas Instrument's TMS320C40, and Motorola's 96002. The design of each of these chips reflect tradeoffs between maximizing performance on the one hand while attempting to maximize the number of problems that they can address effectively on the other. An example of a highly specialized architecture for several cyclostationary algorithms may be found in Roberts [Ref 5].

## **B. BASIC ARCHITECTURAL CONCEPTS**

The architectures described below represent an additional level of specialization, relative to the DSP chips noted above, although not as specialized as the application-specific architectures described in Roberts [Ref 5]. These architectures are limited to vector operations such as vector multiply or add, radix- $r$  butterflies, and the dot product of two vectors. The most distinguishing feature of architectures incorporating BFM's is that a single operation type (e.g., radix-2 butterfly) is performed on a block of data. Further, they are fully pipelined such that any operation can be completed in the same number of cycles as there are resultants to be stored plus latency.

A typical BFM architecture is shown in Figure III.2. For each pass, the BFM is initialized with an operation code (op code) and data flow information. Data is then streamed through the BFM from an input buffer to an output buffer. The op code specifies the particular operation to be performed on the data. *Address generators* (AGs) are necessary for each buffer to ensure that the proper data is passed at the appropriate time. The AGs receive control signals from the controller which decodes the flow control code to produce these signals. Given that the memories can service references at the clock speed of the processor, the vectors can be processed efficiently.

There are however, two sources of conflicts that can diminish the efficiency of this highly pipelined architecture: memory conflicts and processor conflicts. The timing diagram of Figure III.3a illustrates a vector processor that flushes the processor pipeline

prior to beginning a new pass. This flushing time is equal to the latency of the operation. In Figure III.3a,  $D$  cycles are required to flush the pipeline, each requiring  $T_c$  seconds. The situation where a processor does not have sufficient resources to begin a new operation without completing the previous operation is called a processor conflict. Figure III.3b illustrates the performance of a vector processor that can operate without processor conflicts.

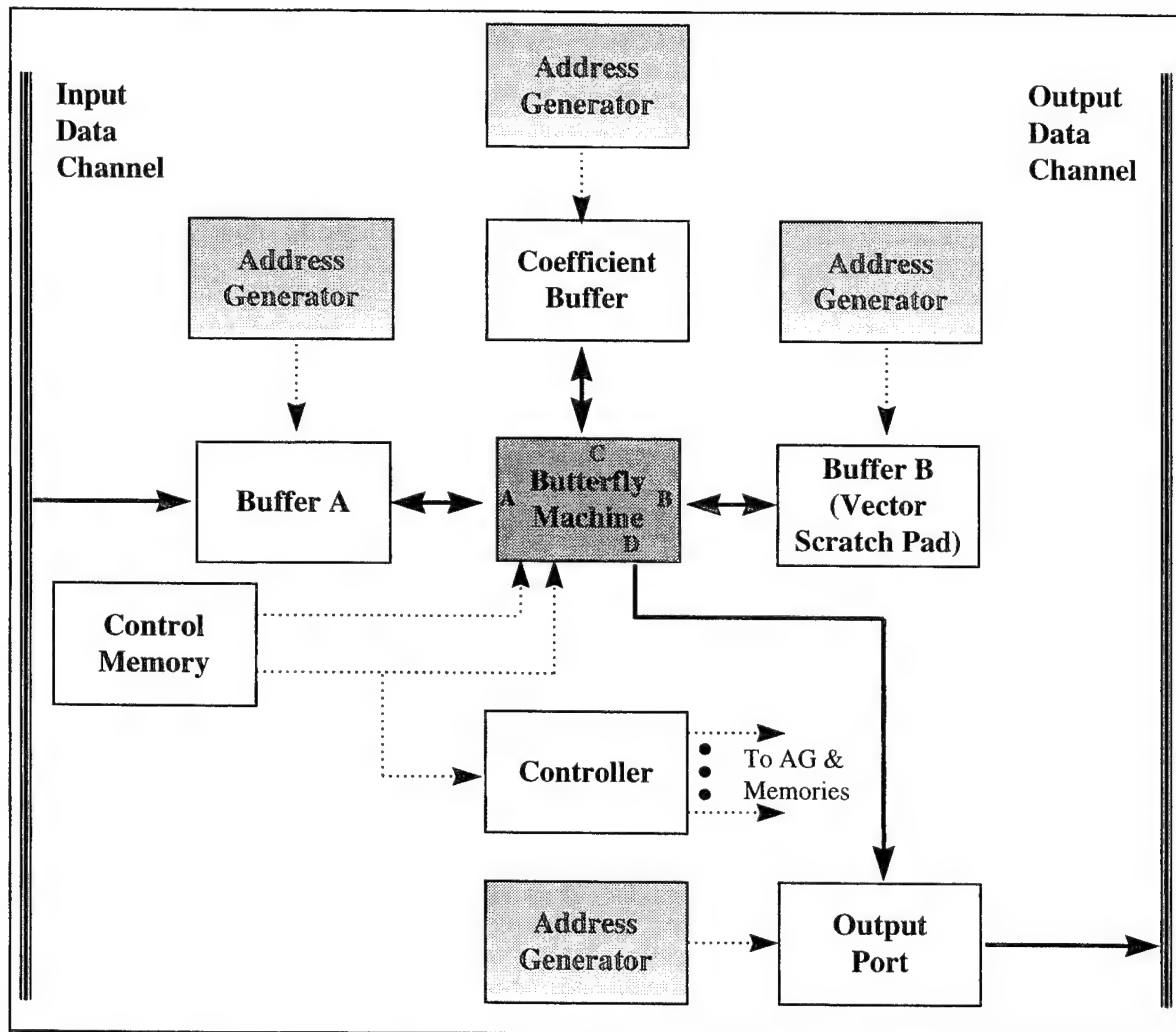
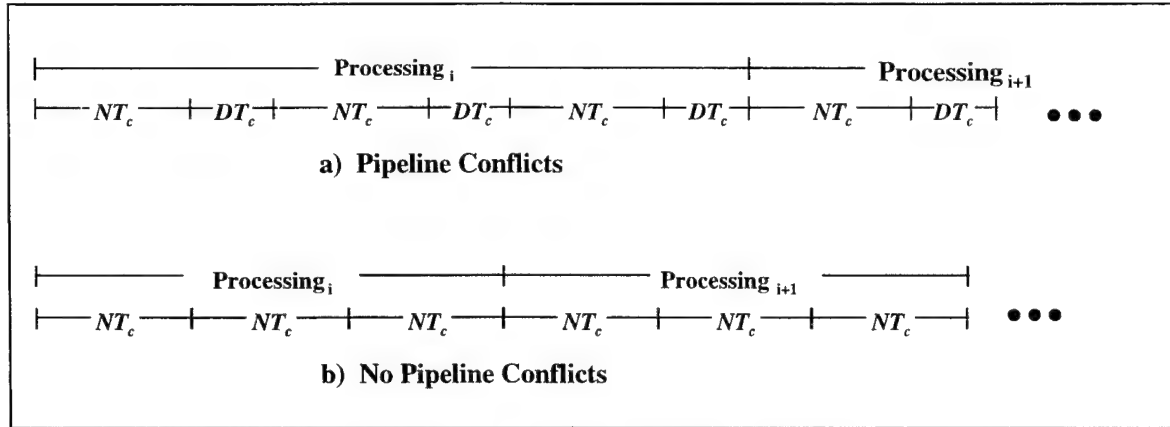


Figure III.2 General Vector Machine Architecture



**Figure III.3 Vector Timing Diagram**

Suppose that a processor contains sufficient resources so that no conflicts will occur. In order to process data as shown in Figure III.3b, the memory system must provide data to the processor at the appropriate time. The situation where the memory system fails to process memory references at the rate required by the processor is called a memory conflict. Therefore, conflict-free operations occur only when both the processor has sufficient resources to avoid flushing the pipeline between operations, and when the memory system can process memory references as the processor requests them.

The effect of conflicts on performance is illustrated in Figure III.3. Each pass has associated with it a vector of length  $N$ , and an operation with an associated latency of  $D$  cycles. Efficiency of the pipeline can be expressed as the ratio of the number of cycles required with no latency and the number of cycles actually required for a given operation

$$E = \frac{N}{N + D}. \quad (\text{III.1})$$

The value  $N$  is a function of the problem domain whereas  $D$  is a characteristic of the implementation of the processor. The efficiency is clearly related to the ratio of  $N$  and  $D$ . A reasonable range of  $D$  is from 10 to 60 where 60 represents the latency for a radix-16 operation. Cyclostationary algorithms generally operate on large data sets as large as  $2^{20}$  or greater. The loss in efficiency is low and the corresponding simplicity in design is significant in both the processor and memory design when the pipeline is flushed between each pass.

The program is loaded into the control memory through the input data channel. Vectors of data are then sent to the DSP architecture through the input data channel. Each vector is processed using one or more passes, and then sent to the output data channel through the output port. This basic architecture and the concept of a four port device in particular, is borrowed from Array Microsystems [Ref 44] and Sharp [Ref 45].

The heart of a program for the BFM is a list of passes. A pass results in streaming a set of data from one buffer to another through the BFM, performing some operation. A pass is defined as shown in Table III.1. A block, the organizational unit for the BFM software, consists of a list of passes plus an input specification to indicate the origin of data for the first pass, and an output specification that states where to send the resulting data. Programs are constructed by stringing blocks together and through the use of super-blocks.

pass := source(s), destination, op code	
source:	buffer id
	base address
	address sequence type
	port id
destination:	buffer id
	base address
	address sequence type
	port id

**Table III.1 Pass Definition**

To illustrate a simple use of an architecture incorporating BFMs, consider the computation of a 1024 point ( $2^{10}$ ) FFT with the architecture illustrated in Figure III.2. Assuming that radix-2 and 16 butterflies are available in the BFM, the FFT may be computed by performing one radix-2 and two radix-16 butterfly passes for a total of three passes on the data. The definition of the passes for this example is contained in



Table III.2 and illustrated in Figure III.4 through Figure III.6. In Figure III.4, the operation to be performed is a radix-2 butterfly beginning with data in buffer A. The input vector enters the processor through port A, is streamed through the processor, and stored in buffer B. The weighting factors are supplied from the coefficient buffer through port C. The second pass, illustrated in Figure III.5, has a radix-16 operation with data now in buffer B and streamed back to buffer A. The coefficient buffer serves in an analogous role but for radix-16 weighting factors. A second radix-16 pass is executed in pass three to complete the  $2^{10}$  point FFT as shown in Figure III.6. The destination buffer is the output port for this pass.

Pass #		
1	Source(s)	Buf_A0,0,bit_rev,port_A
		Buf_Coef,1024,radix2,port_C
	Destination	Buf_B,0,linear,port_B
2	Source(s)	Buf_B0,0,const_geo,port_B
		Buf_Coef,0,radix16,port_C
	Destination	Buf_A,0,linear,port_A
3	Source(s)	Buf_A0,0,const_geo,port_A
		Buf_Coef,1024,radix16,port_C
	Destination	Buf_Out,0,linear,port_B

**Table III.2 Pass Description**

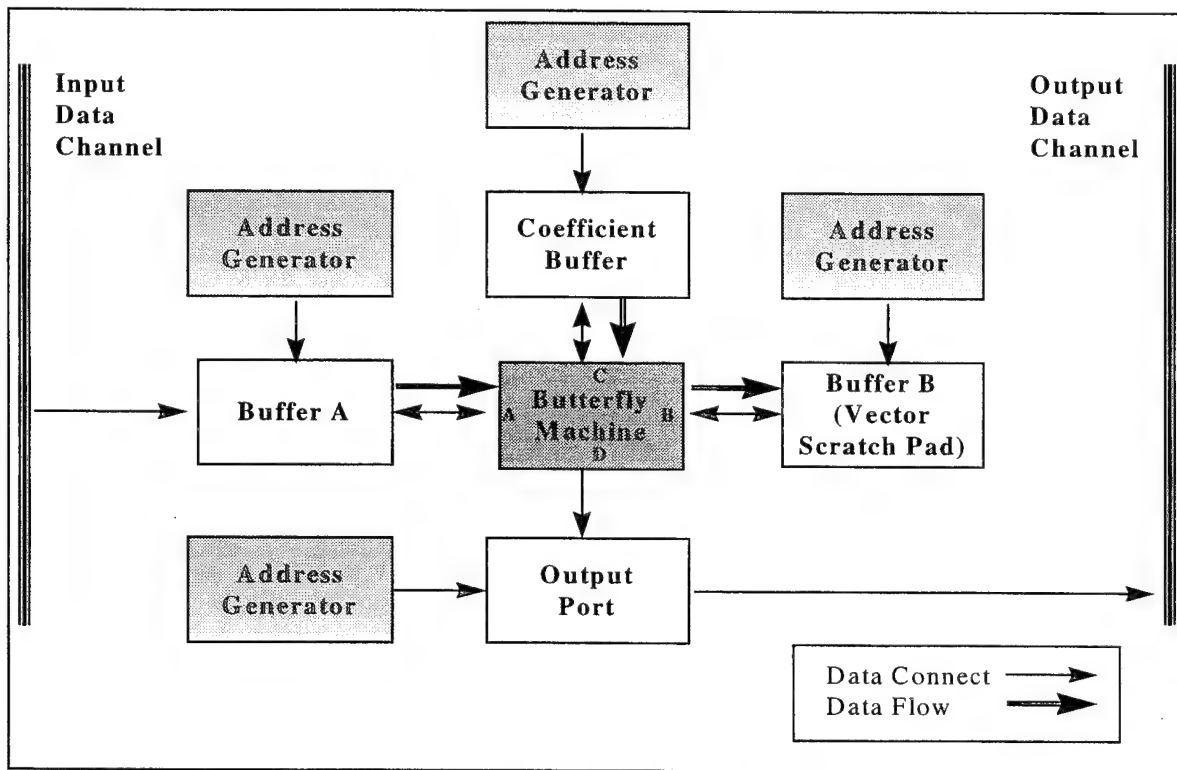


Figure III.4 1024-Point FFT: Pass 1

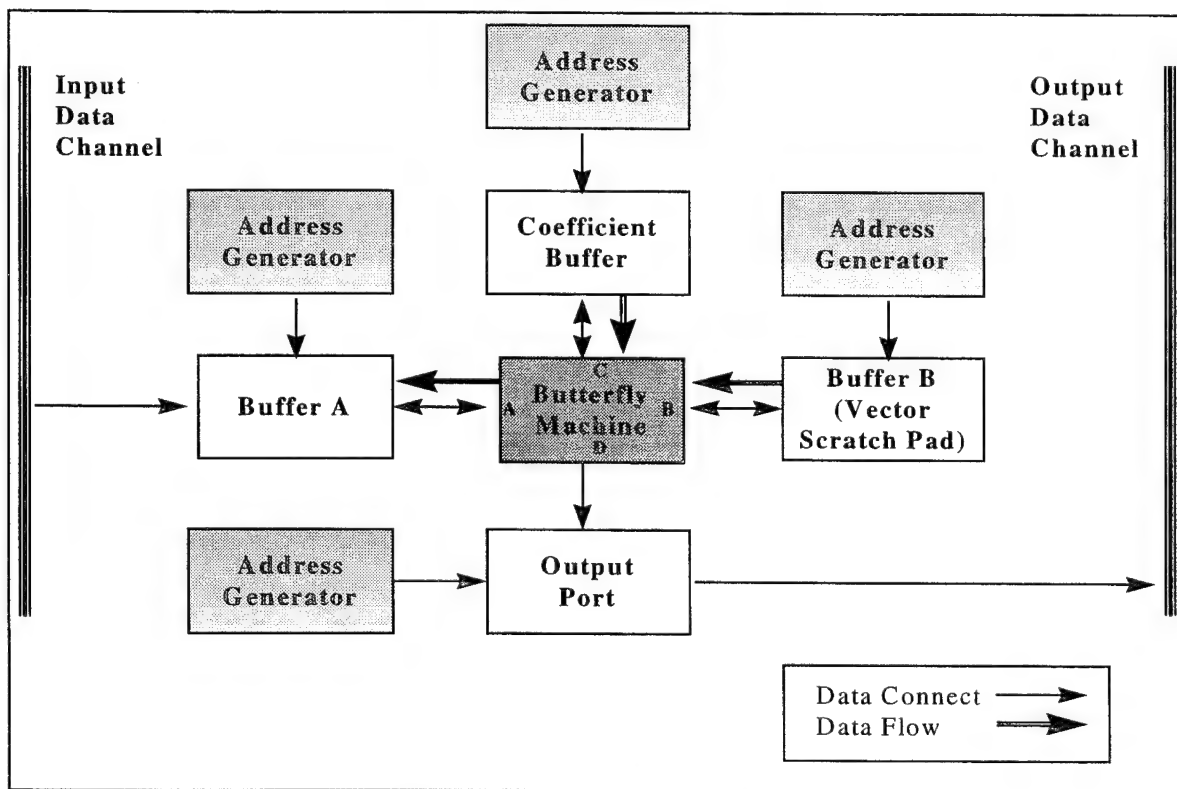


Figure III.5 1024-Point FFT: Pass 2

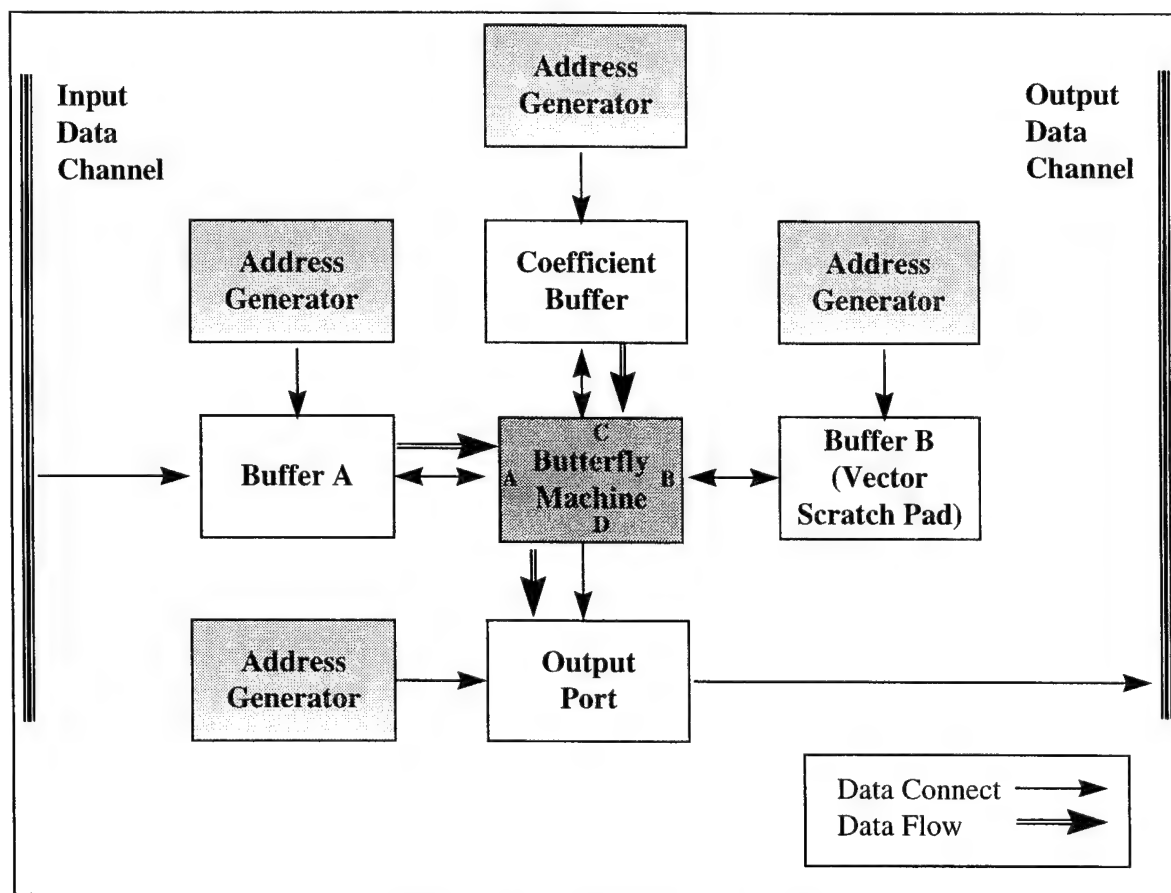


Figure III.6 1024-Point FFT: Pass 3

A timing diagram for the 1024 point FFT is shown in Figure III.7. Note that  $N$  is the number of elements in the vector.  $D_2$  and  $D_{16}$  are the latencies associated with the radix-2 and 16 operations respectively. Notice that it indicates that input, and output can be overlapped with processing keeping the processor fully utilized. This is possible by

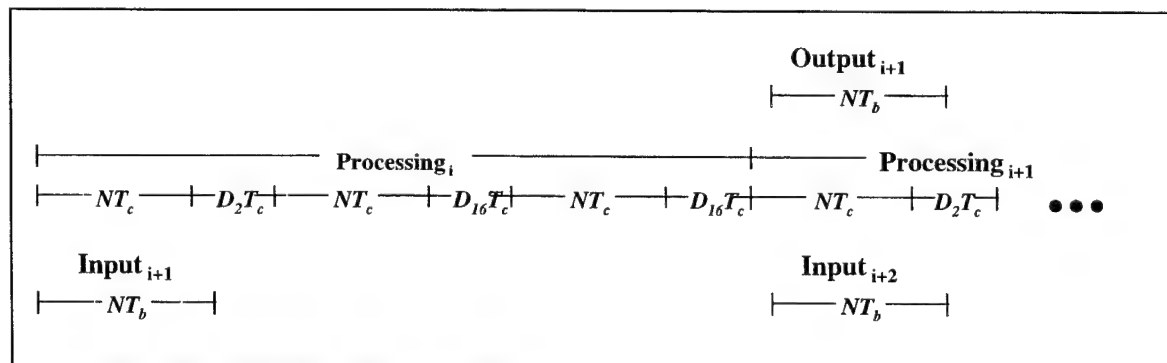


Figure III.7 Timing Characteristics for 1024-Point FFT

carefully selecting the ordering of buffers between which the data is passed. The selection depends upon whether there will be an odd or even number of passes. Additionally, buffer A must be dual ported in order to provide the overlap of input, processing, and output indicated in Figure III.2. This will be discussed further with the one-chip architecture.

### C. PERFORMANCE MEASURES

There are several performance measures that are appropriate to consider when discussing BFMs. *Factor of real time*,  $F_T$ , is defined as the ratio of computation time to collect time and represents the percentage of data that can be processed given that data is collected continuously. [Ref 48]

$$F_T = \frac{\text{Computation Time}}{\text{Collect Time}} = \frac{\sum_u \frac{C_u}{p_{hu}} T_{cu}}{NT_s}, \quad (\text{III.2})$$

where

$C_u$  is the number of computations for hardware type  $u$ ,

$p_{hu}$  is the number of hardware units of hardware type  $u$ ,

$T_c$  is the clock interval,

$N$  is the number of samples taken, and

$T_s$  is the sample interval.

This reduces to

$$F_T = \frac{CT_c}{Np_hT_s}, \quad (\text{III.3})$$

when there is only one computational hardware resource type as is the case for BFMs. The computation time for BFMs is defined as the sum of the product of the number of passes and the pass length, for each type of operation.

Efficiency of a parallel architecture is defined as

$$E_k = \frac{C_{Req\ k}}{C_{Used\ k}} \quad (\text{III.4})$$

where

$$C_{Req\ k} = \frac{C_{Req\ 1}}{k}, \quad (III.5)$$

and  $C_{Req\ 1}$  is the number of processing cycles required to compute a function with a single BFM and  $C_{Used}$  is the number of cycles actually used by a  $k$ -processor BFM.

#### D. FAST FOURIER TRANSFORM

The set of fast Fourier transform (FFT) algorithms selected for the butterfly machine architecture is those that are developed using *decimation-in-frequency*. The butterfly machine architecture includes radix-2, 4, and 16 butterflies. Butterflies with radices with powers of two have been selected for their efficient implementation gained through algorithmic techniques. Further, the radix-4 butterfly has a straightforward hardware implementation due to the fact that the complex exponential takes on values of  $\pm 1$  and  $\pm j$ , which allows the use of hardware addition in place of multiplication in some cases. Butterflies of radix-2 are supported to allow FFTs of any vector of length  $2^k$ .

Two early works concerning implementation of the FFT can be found in Singleton [Ref 49] and Pease [Ref 50]. The decimation-in-frequency algorithm discussed below is described in Oppenheim [Ref 51] for a radix-2 butterfly. Figure III.8 is a signal flow graph for the decimation-in-frequency algorithm for an eight point vector. Although this algorithm can be implemented for the butterfly machine architecture, the addressing reference stream causes two problems. First, the radix-2 butterfly pattern varies from pass to pass. In the first radix-2 pass, the butterfly indices are separated by four (e.g.,  $x(0)$  and  $x(4)$ ). In the second and third passes, the indices are separated by two and one respectively. This variation in address patterns must take into account by the address generators (see Figure III.2). Second, the analysis of memory performance is made more difficult by the address pattern changes from pass to pass.

Both problems are simplified by replacing the in-place signal flow graph with a constant geometry signal flow graph. The corresponding constant geometry signal flow graph for Figure III.8 is shown in Figure III.9.

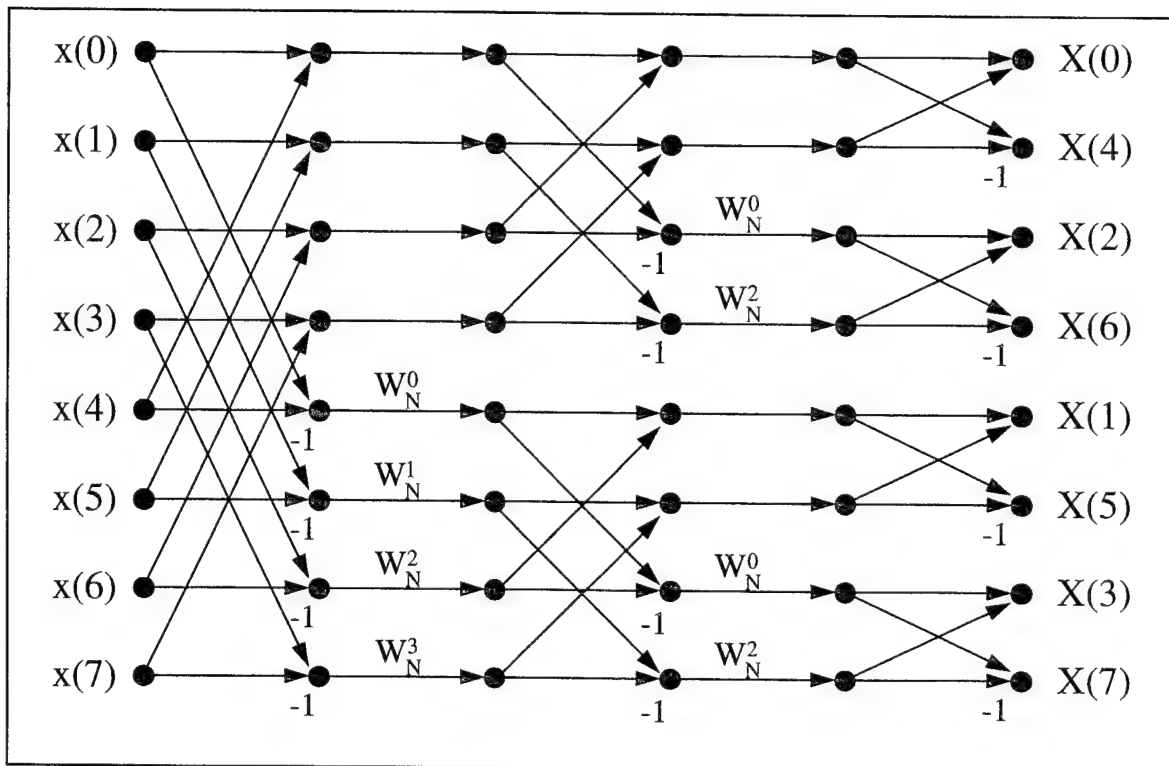


Figure III.8 Radix-2 In-place Decimation-in-frequency Flow Graph

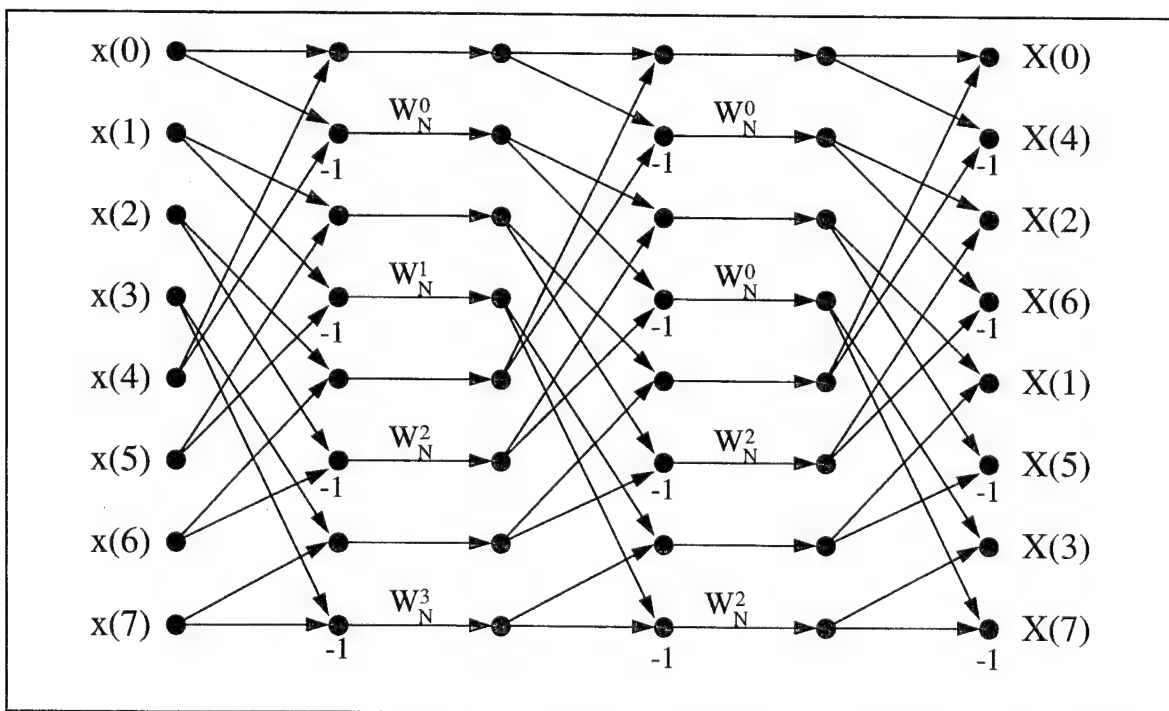


Figure III.9 Radix-2 Constant-Geometry Decimation-in-frequency Flow Graph

The following demonstrates decomposition with decimation-in-frequency for a radix-4 butterfly. For a sequence

$$x[n] \quad n = 0, 1, \dots, N-1, \quad (\text{III.6})$$

the *Discrete Fourier Transform* (DFT) is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \quad k = 0, 1, \dots, N-1 \quad (\text{III.7})$$

where

$$W_N = e^{-j\frac{2\pi}{N}}. \quad (\text{III.8})$$

The sequence  $x[n]$  is partitioned into the number of sets equal to the radix number. For a radix-4 butterfly, Equation (III.7) becomes:

$$X[k] = \sum_{n=0}^{N/4-1} x[n] W_N^{nk} + \sum_{n=N/4}^{N/2-1} x[n] W_N^{nk} + \sum_{n=N/2}^{3N/4-1} x[n] W_N^{nk} + \sum_{n=3N/4}^{N-1} x[n] W_N^{nk}. \quad (\text{III.9})$$

A change of variables in the second, third, and forth summations yields

$$X[k] = \sum_{n=0}^{N/4-1} x[n] W_N^{nk} + \sum_{n=0}^{N/4-1} x[n + \frac{N}{4}] W_N^{(n+\frac{N}{4})k} + \sum_{n=0}^{N/4-1} x[n + \frac{N}{2}] W_N^{(n+\frac{N}{2})k} + \sum_{n=0}^{N/4-1} x[n + \frac{3N}{4}] W_N^{(n+\frac{3N}{4})k}. \quad (\text{III.10})$$

Moving the parts of the weighting factors that are not dependent on the summations and using Equation (III.8) yields

$$X[k] = \sum_{n=0}^{N/4-1} x[n] W_N^{nk} + W_4^k \sum_{n=0}^{N/4-1} x[n + \frac{N}{4}] W_N^{nk} + W_2^k \sum_{n=0}^{N/4-1} x[n + \frac{N}{2}] W_N^{nk} + W_4^{3k} \sum_{n=0}^{N/4-1} x[n + \frac{3N}{4}] W_N^{nk}. \quad (\text{III.11})$$

Consider the following four sets of  $X[k]$  such that  $k = 4r$ ,  $k = 4r+1$ ,  $k = 4r+2$ , and  $k = 4r+3$  for  $r = 0, 1, \dots, \frac{N}{4}-1$ . Substituting these values of  $k$  into Equation (III.11) and again using Equation (III.8) yields the following four equations for  $X[k]$ :

$$X[4r] = \sum_{n=0}^{N/4-1} \left[ x[n] + x\left[n + \frac{N}{4}\right] + x\left[n + \frac{N}{2}\right] + x\left[n + \frac{3N}{4}\right] \right] W_{N/4}^{nr} \quad (\text{III.12})$$

$$X[4r+1] = \sum_{n=0}^{N/4-1} \left[ x[n] - jx\left[n + \frac{N}{4}\right] - x\left[n + \frac{N}{2}\right] + jx\left[n + \frac{3N}{4}\right] \right] W_N^n W_{N/4}^{nr} \quad (\text{III.13})$$

$$X[4r+2] = \sum_{n=0}^{N/4-1} \left[ x[n] - x\left[n + \frac{N}{4}\right] + x\left[n + \frac{N}{2}\right] - x\left[n + \frac{3N}{4}\right] \right] W_N^{2n} W_{N/4}^{nr} \quad (\text{III.14})$$

$$X[4r+3] = \sum_{n=0}^{N/4-1} \left[ x[n] + jx\left[n + \frac{N}{4}\right] - x\left[n + \frac{N}{2}\right] - jx\left[n + \frac{3N}{4}\right] \right] W_N^{3n} W_{N/4}^{nr} \quad (\text{III.15})$$

Figure III.10 illustrates the use of Equations(III.12) through (III.15) to compute in part, an FFT using a radix-4 butterfly. The eight point vector is passed through a radix-4 followed by a radix-2 butterfly. The second radix-4 butterfly is not shown for clarity. The constant-geometry version is constructed in an analogous manner as the radix-2 version shown above.

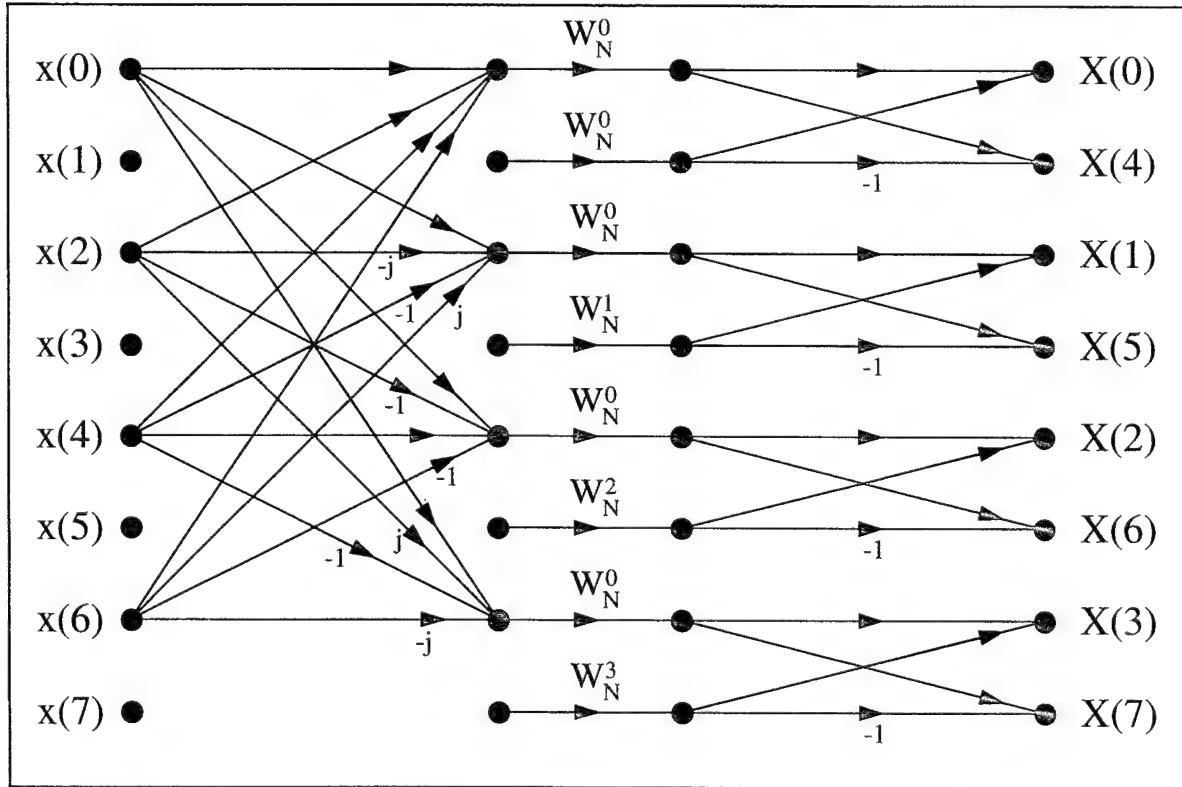


Figure III.10 Radix-4/Radix-2 In-place Decimation-in-frequency Flow Graph



## E. PERMUTATION-BASED MEMORY DECODING SCHEME

The memory decoding scheme selected for the butterfly machine architecture is permutation based, as described in Section D of Chapter II. This selection is based on finding the least expensive implementation that provides excellent throughput for the memory system. Conventional memory decoding performs poorly for algorithms that have a characteristic of powers of two. These algorithms include radix- $r$  butterflies where  $r=2^k$  and the digit-reversed patterns which are required for the last pass of the FFT, as described in Section D above. The 1-Skew and prime number memory decoding schemes are more complex to implement than the permutation-based method. Also, most prime number decoding schemes do not use all of the physical memory, as indicated in Section D of Chapter II.

The following discussion of permutation-based memory decoding will first describe a set of constraints necessary to construct a memory decoding scheme that yields a valid interleaved memory system. Then, a set of constraints will be described that yields the desirable properties for a memory used in the butterfly machine architecture. A specific permutation matrix will then be constructed that is designed for the butterfly machine architecture.

First, terminology concerning permutation-based memory decoding will be established. The address space contains  $2^N$  words indexed  $0 \dots 2^N - 1$ . A binary address is written  $a_{N-1}a_{N-2} \dots a_1a_0$  where the most significant bit (MSB) of the address has an index of  $N-1$  and the least significant bit has an index of 0. An interleaved memory system contains  $B$  banks, where each bank contains a total of  $K$  words indexed in the conventional manner 0 through  $K-1$ . Therefore, the number of bits required to specify a bank number is

$$n = \log_2(B) \quad (\text{III.16})$$

and the number of bits required to specify the index into a memory bank is

$$k = \log_2(K). \quad (\text{III.17})$$

The number of bits for the memory address space is then

$$R = n + k. \quad (\text{III.18})$$

The memory decoding scheme must specify a bank number and an index into the bank. This index will be referred to as the bank index. In general, the bank number is specified with the following matrix equation:

$$\begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_0 \end{bmatrix} = \begin{bmatrix} p_{0,0} & \cdots & p_{0,R-n} & \cdots & p_{0,r-1} \\ p_{1,0} & \cdots & p_{1,R-n} & \cdots & p_{1,r-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{n-1,0} & \cdots & p_{n-1,R-n} & \cdots & p_{n-1,r-1} \end{bmatrix} \begin{bmatrix} a_{r-1} \\ a_{r-2} \\ \vdots \\ a_0 \end{bmatrix}. \quad (\text{III.19})$$

or

$$\mathbf{b} = \mathbf{P} \cdot \mathbf{a} \quad (\text{III.20})$$

when shorthand notation is appropriate.

The resulting vector  $\mathbf{b}$  is a binary representation of the bank number. The vector  $\mathbf{a}$  represents the  $r$  LSBs of the address used to decode the bank number. Note that

$$n \leq r \leq R. \quad (\text{III.21})$$

Entries in the  $\mathbf{P}$  matrix are either a **1** or a **0**. A bit  $b_i$  of the bank number is a result of the normal dot product of the  $i$ th row of matrix  $\mathbf{P}$  and the  $\mathbf{a}$  column address vector except that the multiplications are logical ANDs and the summation is a logical exclusive OR. The  $i$ th bit of the bank number can be written as

$$b_i = (p_{i,0} \cdot a_{k-1}) \oplus (p_{i,1} \cdot a_{k-2}) \oplus \cdots \oplus (p_{i,r-2} \cdot a_1) \oplus (p_{i,r-1} \cdot a_0) \quad (\text{III.22})$$

revealing that each bit of the bank number can be thought of as an encoding based on the parity of selected bits of the address as determined by the  $\mathbf{P}$  matrix entries that are equal to **1**.

To verify that permutation-based memory decoding provides a valid memory map, it will first be shown that conventional memory decoding, which is a valid memory map by inspection, is a subset of permutation-based memory decoding. Then, variations of

this equivalent conventional memory decoding scheme will be explored to determine the constraints on the permutation matrix that are required to ensure a valid memory map.

As an initial point of reference for analysis of permutation memory decoding, note that if the permutation matrix  $\mathbf{P}$  is the identity matrix with dimensions  $n$  by  $n$ , then the resulting permutation-based scheme is equivalent to conventional memory decoding. Further, the bank index is computed by directly using the most significant  $k=R-n$  bits. This decoding scheme for the bank index is assumed for the remainder of the document.

The permutation equation for computing the bank number, which is equivalent to conventional decoding, is shown below for a bank number with  $n$  bits.

$$\begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{bmatrix} = \mathbf{I}_{n \times n} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} \quad (\text{III.23})$$

It is useful to organize the linear address space into equally-sized  $2^n$  blocks, where each block begins at  $l \cdot 2^n$  for  $l = 0 \dots 2^{R-n} - 1$ . For conventional memory decoding, the  $k$  MSBs specify a memory location within a bank and the  $n$  LSBs specify the bank as indicated in Equations (III.16) through (III.18). Therefore, for conventional decoding for an arbitrary fixed index, the sequence  $0 \dots 2^n - 1$  on the  $n$  LSBs maps one-to-one and onto the set of bank numbers. This sequence  $0 \dots 2^n - 1$  will be referred to as the *base sequence*. Since this one-to-one mapping is valid for all blocks, the mapping from the linear address space to the bank number, bank index pair space is also one-to-one and onto. The practical implication is that all of the capacity of the memory hardware is utilized and the decoding scheme is valid for a memory system.

Now, consider any change to the  $\mathbf{P}$  matrix specified above such that the dimension of  $\mathbf{P}$  is unchanged and the nonsingularity of the matrix is maintained. A modified but nonsingular matrix  $\mathbf{P}$  of equal dimension will yield a different mapping (i.e., it will not be the identity mapping), but will still map the base sequence one-to-one

and onto for each block. Therefore, any such matrix  $\mathbf{P}$  will produce a valid memory that utilizes all of the memory. Also, any such matrix  $\mathbf{P}$  generates the desired round robin pattern for a sequential memory reference stream for an interleaved memory system since each bank is selected exactly once for each base sequence.

Now consider a matrix  $\mathbf{P}$  of dimension  $n$  by  $R$  (i.e., use possibly all of the address bits to generate the bank number) which is constructed by concatenating columns to the left of matrix  $\mathbf{P}$ , described in the previous paragraph. All of the values in each of the new columns are assumed to be  $0$  except for a single  $1$  in an arbitrary  $i$ th row and  $j$ th column as shown in Equation (III.24).

$$\begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \vdots & \cdots & 0 & \cdots & \vdots \\ 0 & 0 & 1_{i,j} & 0 & 0 \\ \vdots & \cdots & 0 & \cdots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{I}_{n \times n} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} \quad (\text{III.24})$$

The identity matrix is used for illustration in Equation (III.24) however, the comments that follow apply equally to any matrix  $\mathbf{P}$  where the identity sub-matrix is replaced with a nonsingular sub-matrix of dimension  $n$  by  $n$ .

Consider the effect of  $p_{i,j} = 1$  on bank selection for a base sequence. The address bit  $a_j$  that corresponds to  $p_{i,j}$  in the  $\mathbf{P}$  matrix is for a given address, either a  $1$  or a  $0$ . When it is a  $0$ , it has no effect on the bank selection (i.e., the bank number is the same number that would have been computed if  $p_{i,j} = 0$ ). When the address bit  $a_j = 1$ , the  $i$ th bank bit,  $b_i$ , is complemented from what it was when  $a_j = 0$  (or when  $p_{i,j} = 0$ ). This results in a permutation of the bank numbers generated for a base sequence. This permutation can be illustrated by listing the original bank numbers generated when the address bit  $a_j = 0$  as a table. Each row in the table is a bank number and the columns represent bit positions for the bank numbers. The permuted set of bank numbers is found by complementing the  $i$ th column of the table. Clearly the new map is still one-to-one and

onto for each base sequence. Since this is valid for all blocks, the mapping represented by the bank decoding scheme indicated in Equation (III.24) is one-to-one and onto.

Recall that the element of the  $\mathbf{P}$  matrix  $p_{i,j} = 1$  was selected for an arbitrary  $i$  and  $j$ . Further, once the mapping resulting from adding a 1 at  $p_{i,j}$  is established to be a one-to-one and onto mapping, the  $\mathbf{P}$  matrix can be modified again by selecting another element  $p_{i,j} = 1$ . The  $\mathbf{P}$  matrix continues to provide a one-to-one and onto mapping based on the rational used for  $p_{i,j}$ . Other elements on the left-hand side of the  $\mathbf{P}$  matrix can be modified as desired while maintaining a memory decoding scheme that is one-to-one and onto. In summary, it can be seen that as long as a nonsingular sub-matrix of dimension  $n$  by  $n$  is positioned in the far right columns of the permutation matrix  $\mathbf{P}$ , other elements of the matrix  $\mathbf{P}$  can be modified in an arbitrary fashion while maintaining a valid memory decoding scheme which also utilizes all of the physical memory.

The next discussion will describe a set of constraints that ensure that a permutation matrix will provide maximum throughput for address patterns with constant stride  $s$  such that

$$s = 2^v \quad (\text{III.25})$$

where  $v$  is an integer, greater than or equal to zero.

As indicated above, Equation (III.23) generates a round robin pattern of bank numbers by selecting each bank once within a base. Also note that the base sequence is an address sequence of constant stride of one. This is true for any matrix  $\mathbf{P}$  such that the sub-matrix  $\mathbf{I}_{n \times n}$  has dimension  $n$  by  $n$  and is nonsingular.

Now consider Table III.3. Table III.3 contains the decimal and binary representation of the counting sequence 0... 15. It is easily verified that the value of the LSB ( $b_0$  in Table III.3) does not change for a sequence with constant stride of two. Further, if the  $b_0$  column of bits is removed from the counting sequence generated with a constant stride of two and the resulting columns are relabeled such that  $b_i$  is labeled  $b_{i-1}$ , for each  $i$ , then the result is a sequence equivalent to the original sequence (i.e., a

sequence with constant stride of one). Therefore, the LSB does not contribute to the selection of a bank when the stride is two and the resulting sequence is a sequence with stride of one when ignoring the column of the least significant bit. A similar set of statements can be made for other strides of powers of two. In general, a stride  $s$  such that

$$s = 2^k \quad k > 0 \text{ an integer} \quad (\text{III.26})$$

will not engage the  $k$ th LSBs and the resulting pattern when removing the  $k$  right most columns will yield a sequence with stride of one.

Using Equation (III.24) as a point of reference, it is desired to modify the  $\mathbf{P}$  matrix such that address patterns with strides of two map to all of the banks within a base sequence. It is clear that if the identity sub-matrix in Equation (III.24) is shifted one position to the left, then the base sequence would map directly into the bank numbers, as is the case for a base sequence in Equation (III.23). Further, if the shifted identity sub-matrix were modified, but its order maintained and remained nonsingular, then the base sequence would continue to map one-to-one and onto the banks. This would naturally destroy the desired pattern for the constant stride of one. Therefore, the task is to find modifications to the matrix  $\mathbf{P}$  that will preserve the desirable properties of a constant stride of one while enhancing the matrix  $\mathbf{P}$  to accommodate address patterns with constant stride of two. In general, the objective is to find a technique for enhancing a matrix  $\mathbf{P}$  that can accommodate constant strides up to a stride of  $2^i$ , such that the matrix can also accommodate strides up to  $2^{i+1}$ .

Consider the matrix Equation (III.27) where the sub-matrix  $\mathbf{P}_{n \times n}^0$  is dimension  $n$  by  $n$  and nonsingular. In the following discussion, sub-matrices  $\mathbf{P}_{n \times n}^v$  for various values of  $v$  will be defined. In all cases, the dimension of these sub-matrices is  $n$  by  $n$  and the subscript will be dropped in the text for brevity.

$$\begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{bmatrix} = \mathbf{0}_{n \times R-n} \quad \vdots \quad \mathbf{P}_{n \times n}^0 \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} \quad (\text{III.27})$$

Decimal	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

**Table III.3 Binary Counting Sequence**

Based on the earlier results, the permutation matrix  $\mathbf{P}$  in Equation (III.27) provides a proper mapping to bank numbers for a valid memory. The sub-matrix  $\mathbf{P}^0$  also provides for a one-to-one mapping of the base sequence, and therefore address patterns, with a stride of one ( $2^0$ ) to bank numbers. It is constructed with the  $R-n-1$  through  $R-1$

columns of the matrix  $\mathbf{P}$ . Consider a new sub-matrix,  $\mathbf{P}^1$  with dimension  $n$  by  $n$ , beginning in column  $R-n-2$  and ending in column  $R-2$  as shown in Equation (III.28).

$$\begin{bmatrix} b_{n-1} \\ b_{n-2} \\ \vdots \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix} \mathbf{0}_{n \times R-n-1} \quad \begin{bmatrix} & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix} \mathbf{P}_{n \times n}^1 \begin{bmatrix} p_{0,n-1}^0 \\ p_{1,n-1}^0 \\ \vdots \\ p_{n-2,n-1}^0 \\ p_{n-1,n-1}^0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_1 \\ a_0 \end{bmatrix} \quad (\text{III.28})$$

None of the values of matrix  $\mathbf{P}$  have changed from Equation (III.27) to Equation (III.28). The sub-matrix  $\mathbf{P}^1$  is defined to be the sub-matrix consisting of columns  $R-n-2$  through  $R-2$  of matrix  $\mathbf{P}$ . Clearly  $\mathbf{P}^1$  is singular since the first column ( $R-n-2$ th column of matrix  $\mathbf{P}$ ) is zero. However, if the first column is modified (i.e., replacing a  $\mathbf{0}$  with a  $\mathbf{1}$  in one or more rows of column  $R-n-2$ ) such that  $\mathbf{P}^1$  is nonsingular, then address patterns of constant stride of two ( $2^1$ ) will map one-to-one and onto the banks.

Consider the effect of replacing a  $\mathbf{0}$  with a  $\mathbf{1}$  in row  $i$ , column  $R-n-2$  of matrix  $\mathbf{P}$ , on an address pattern with a constant stride of one. When the  $a_{i,R-n-2} = 0$ , the base sequence mapping is unchanged. When the  $a_{i,R-n-2} = 1$ , the  $i$ th bit of the bank number is complemented, resulting in a new mapping but a mapping that is still one-to-one and onto. Since  $a_{R-n-2}$  has one more bit of significance than the sub-matrix  $\mathbf{P}_{n \times n}^0$ , one base sequence will be mapped with  $a_{R-n-2} = 0$  followed by one base sequence mapped with  $a_{R-n-2} = 1$ . This pattern will be repeated for address patterns greater than twice the base sequence length.

The process for constructing  $\mathbf{P}^1$  can be repeated for  $\mathbf{P}^2, \mathbf{P}^3, \dots, \mathbf{P}^{R-n}$  such that each sub-matrix  $\mathbf{P}^i$  is nonsingular. Construction of each sub-matrix  $\mathbf{P}^i$  is accomplished as described for  $\mathbf{P}^1$  to support address patterns of constant stride of  $2^i$ .



The effect of constructing  $\mathbf{P}^i$  on address sequences of constant stride of  $2^v$  where  $v = 0 \dots i-1$  will now be described. Construction of  $\mathbf{P}^i$  involves modification of column  $R-n-i-1$  which corresponds to the address bit  $a_{n+i}$ . There are  $2^{n+i}$  addresses where  $a_{n+i} = 0$  followed by  $2^{n+i}$  addresses where  $a_{n+i} = 1$ . This pattern repeats if the address sequence is longer than  $2^{n+i+1}$ . For address patterns with constant stride of one ( $2^0$ ),  $a_{n+i} = 0$  for  $2^i$  base sequences (i.e.,  $2^i$  base sequences are completed while  $a_{n+i}$  is constant). For address patterns with constant stride of two ( $2^1$ ),  $a_{n+i} = 0$  for  $2^{i-1}$  base sequences followed by  $a_{n+i} = 1$  for  $2^{i-1}$  base sequences. In general, for address patterns with constant stride of  $2^v$ ,  $a_{n+i} = 0$  for  $2^{i-v}$  base sequences followed by  $a_{n+i} = 1$  for  $2^{i-v}$  base sequences.

The effect of a set of ones located at  $p_{i,j}$  for various  $i$  and  $j$  in the  $\mathbf{P}$  matrix is cumulative. In general, the row positions dictate the bit position(s) of the bank number to be complimented. The mapping is unique to the row number or set of row numbers (e.g., a one in the  $i$ th row will generate a different map than a one in the  $k$ th row. A one in both the  $i$ th and  $k$ th row is a third mapping). The column number determines how many base sequences will be spanned for a constant value of the address bit  $a_j$ , as described in the previous paragraph.

Figure III.11 and Figure III.12 illustrate the address pattern generated, given the permutation matrix  $\mathbf{P}$  shown below:

$$\mathbf{P} = \left[ \begin{array}{ccc|c} 1_c & 0 & 1_a & \\ 0 & 1_b & 0 & \mathbf{P}^0 \\ 1_c & 0 & 0 & \end{array} \right] \quad (\text{III.29})$$

Figure III.11 illustrates the mapping generated by the nonzero bits on the left-hand side of the dashed line of matrix  $\mathbf{P}$  in Equation (III.29). The top box of Figure III.11 represents the bank pattern resulting from the base sequence given that all of the elements on the left-hand side of the dashed line in Equation (III.29) are zeros. The effect of adding the element labeled  $1_a$  is to first generate the sequence of bank numbers generated

by the base sequence (this occurs when the address bit corresponding to  $1_a$  is zero) alone followed by a permutation of the base sequence labeled mapping #1 in Figure III.11. Adding the element marked  $1_b$  results in a sequence consisting of the base sequence followed by the mapping #1 sequence (address bit corresponding to  $1_b$  is zero) followed by a permutation of the base /mapping #1 sequence referred to as mapping #2 in the figure.

The mapping generated as a result of the two elements in Equation (III.29) labeled  $1_c$  is similar to that described above for  $1_a$  and  $1_b$ . First, the set containing the base sequence concatenated with the mapping #1 sequence, concatenated with mapping #2 is generated. A permuted version of this sequence is then passed and labeled mapping #3.

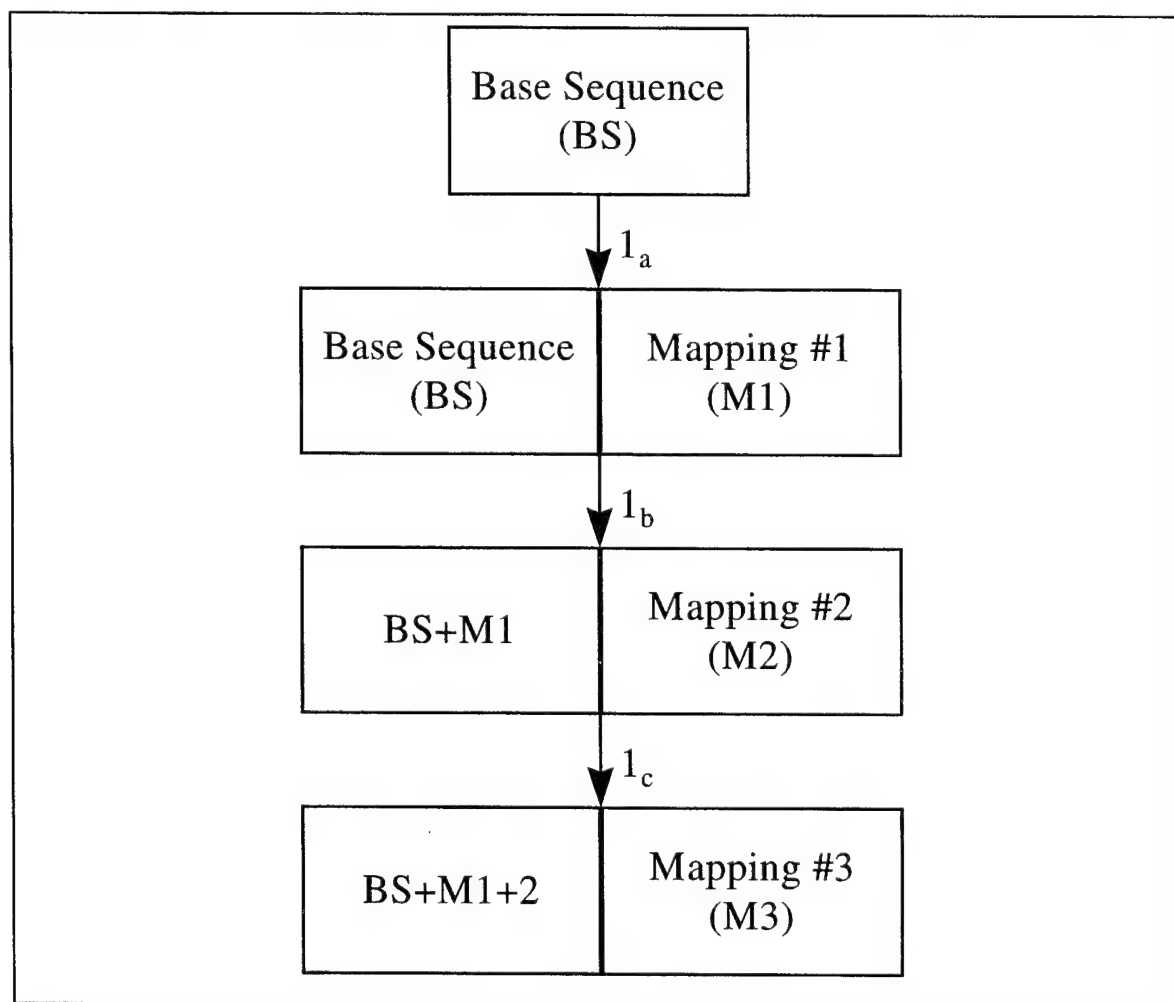
Figure III.12 provides a comparative illustration of the sequences generated by the nonzero elements in Equation (III.29). Figure III.12a) reflects the base sequence pattern of bank numbers, given that all of the sub-matrix on the left-hand side is zero. This results in the repetition of the block of base sequence bank numbers. The bank number pattern shown in Figure III.12b) illustrates the effect of adding  $1_a$  to the matrix. Figure III.12c) and d) reflect the accumulative effect of adding  $1_b$  and  $1_c$  respectively to the bank number pattern.

The simulation runs, based on permutation-based memory decoding described in Chapter II, are based on the following specifications for the interleaved memory system:

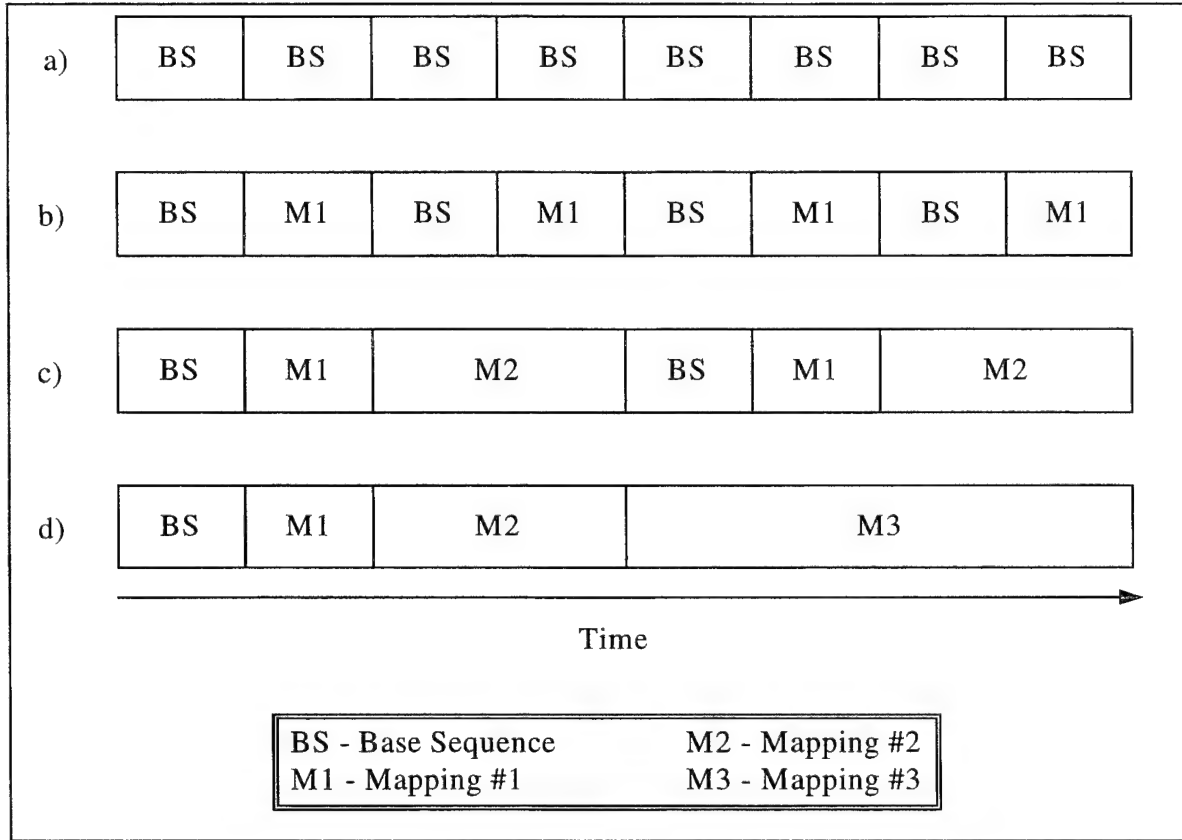
- Number of Banks: 4, 8, 16 and 32.
- Linear Memory Space:  $0 \dots 2^{24} - 1$ .
- The permutation matrices used in the simulations for all address patterns except radix- $r$  butterflies are shown in Figure III.13 through Figure III.16. Permutation matrices for radix- $r$  butterfly patterns are described in Chapter V.

In this section, permutation matrices are described in detail. Requirements sufficient to ensure a valid memory map was described. In particular, if the rightmost  $n$  by  $n$  sub-matrix of the permutation matrix is nonsingular, then the permutation matrix

generates a valid memory mapping. Further, if the  $n$  by  $n$  sub-matrix identified as  $\mathbf{P}^i$  is nonsingular, then address patterns with stride  $= 2^i$  will produce a bank selection pattern that is near ideal for an interleaved memory system. The following section will present the high-level architecture for a single vector processor.



**Figure III.11 Permutation Address Pattern Maps**



**Figure III.12 Comparison of Permutation Address Patterns**

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

**Figure III.13 Simulation Permutation Matrix: NoBanks=4**

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

**Figure III.14 Simulation Permutation Matrix: NoBanks=8**

1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1

**Figure III.15 Simulation Permutation Matrix: NoBanks=16**

0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1

**Figure III.16 Simulation Permutation Matrix: NoBanks=32**

#### F. ONE-CHIP ARCHITECTURE

The one-chip BFM architecture is illustrated in Figure III.17. This architecture consists of a single BFM, six memory buffers, and data multiplexers to control data flow. Not shown explicitly are address generators, necessary for each memory. Buffers A0, A1, B0, and B1, and the auxiliary buffer serve as data sources and destinations. The coefficient buffer contains any constants required by the function executed such as weighting factors for radix- $r$  butterfly operations, windowing data, frequency down conversion data, etc.

Control for accepting data from the input data channel, sending data out onto the output data channel, and computation by the BFM are independent. The basic model for communicating data is message passing. This will be described in more detail when discussing the parallel architecture.

The one-chip BFM will be used to execute the SSCA in the discussion below. The functional diagram for the SSCA is shown in Figure III.18. Three basic blocks are required for the SSCA, namely channelization, correlation multiply, and back-end  $N$  FFT. For illustration purposes, it is assumed that  $N' = 2^5$  and  $N = 2^{17}$ . Channelization will require one pass for windowing, one radix-2 and one radix-16 butterfly pass for the 32-

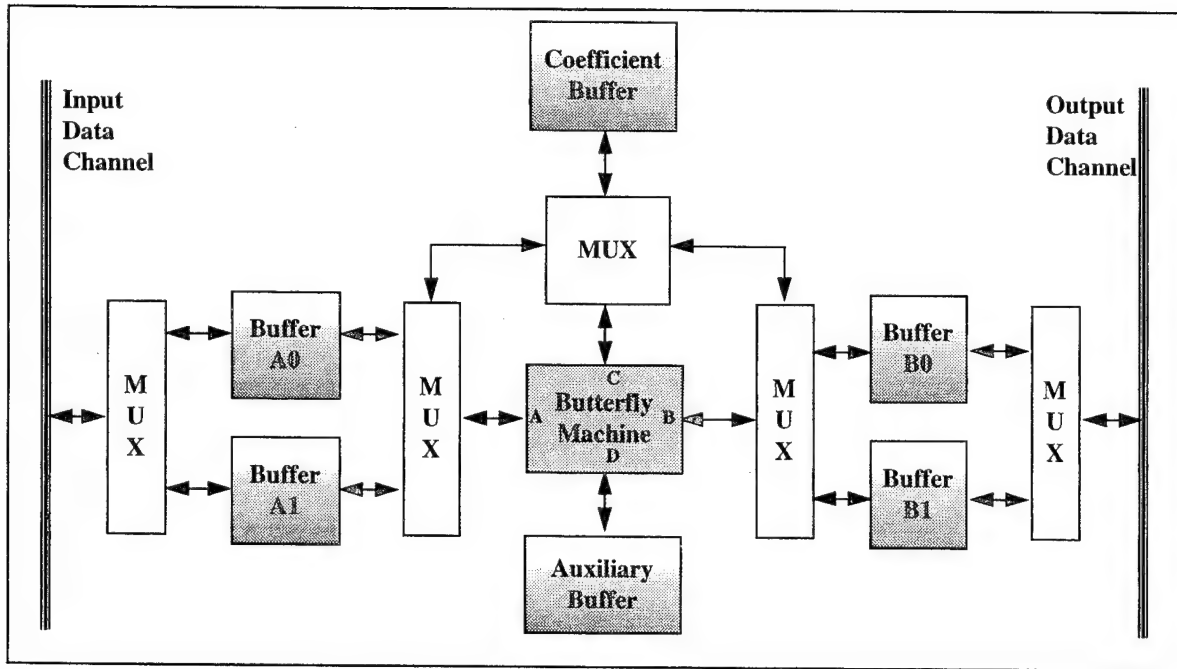


Figure III.17 One-Chip Architecture

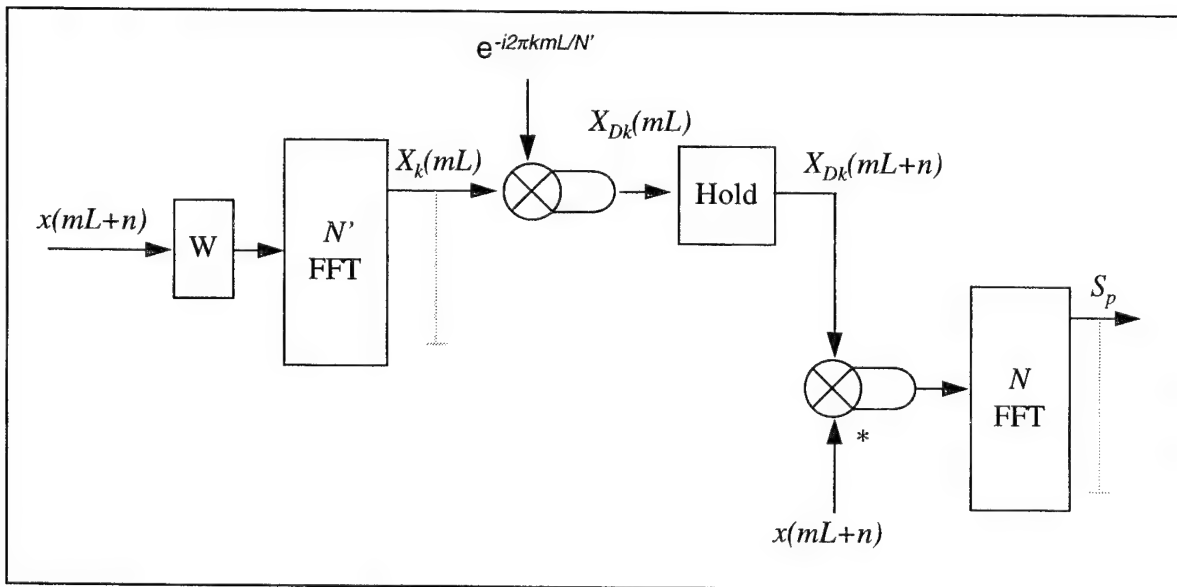
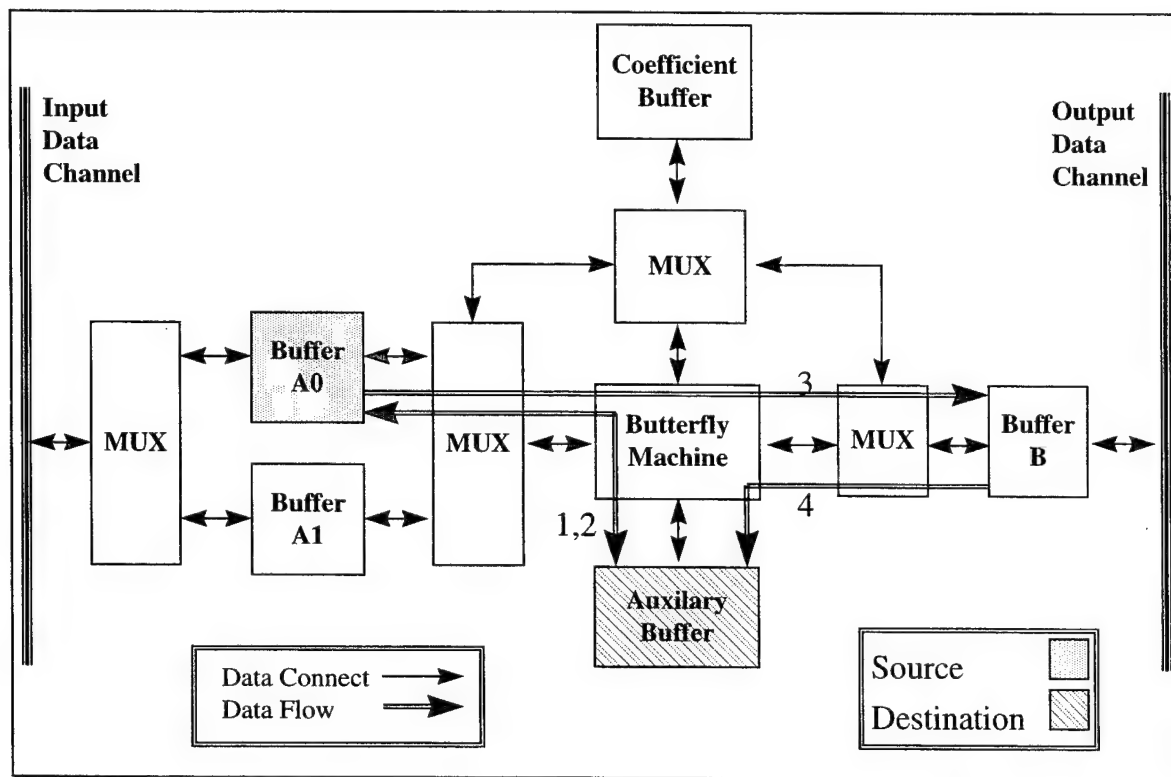


Figure III.18 SSFA Functional Diagram

point  $N'$  FFT, and one pass for down conversion. This channelization block must be performed  $P=4N/N'$  times.

In order to maintain overlap of the input and output with the processing, input will alternate between buffer A0 and buffer A1. Assuming that input data is in buffer A0, data flow for the channelization passes is as shown in Figure III.19 (i.e., data moves from

buffer A0 to the auxiliary buffer and back to buffer A0, then to buffer B and finally to auxiliary buffer). Note that if the number of passes in channelization were odd, the data path would be from buffer A0 to buffer B, and then to the auxiliary buffer.



**Figure III.19 SSCA Execution: Channelization**

The second block, correlation multiply, consists of either a single pass of length  $L$ , the decimation factor, a total of  $P$  times or, a single pass of length  $N$  a total of  $N'$  times. The latter is the method of choice for the one-chip architecture since it results in longer but fewer passes. The former is the better choice for the parallel architecture since these correlation multiplies can be accomplished incrementally by the back-end as each of the  $N'$  samples are passed from the channelizer. The correlation multiply pass is illustrated in Figure III.20. Observe that the original  $N$  data samples are used from buffer A0.

The third block, the back-end  $N$ -point FFT, is computed with one radix-2 and four radix-16 butterfly passes as shown in Figure III.21. Data is ping-ponged between buffer B and the auxiliary buffer such that the final result is located in buffer B.

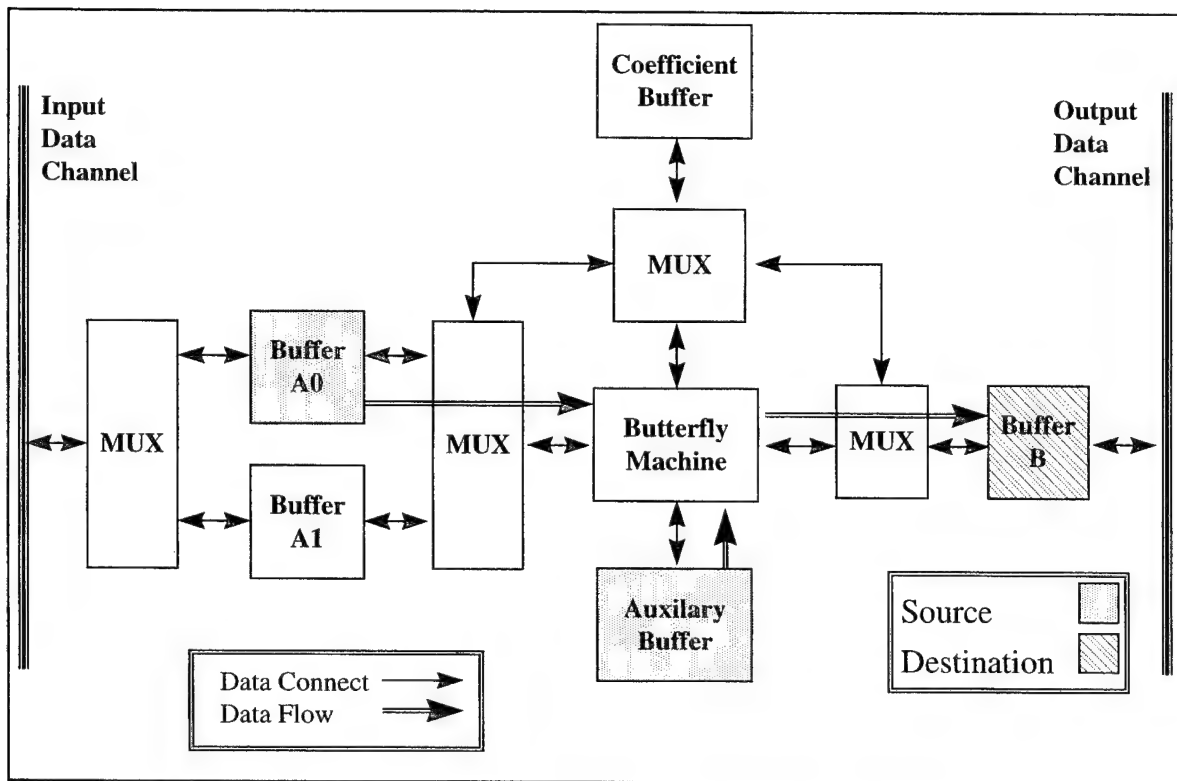


Figure III.20 SSCA Execution: Correlation Multiply

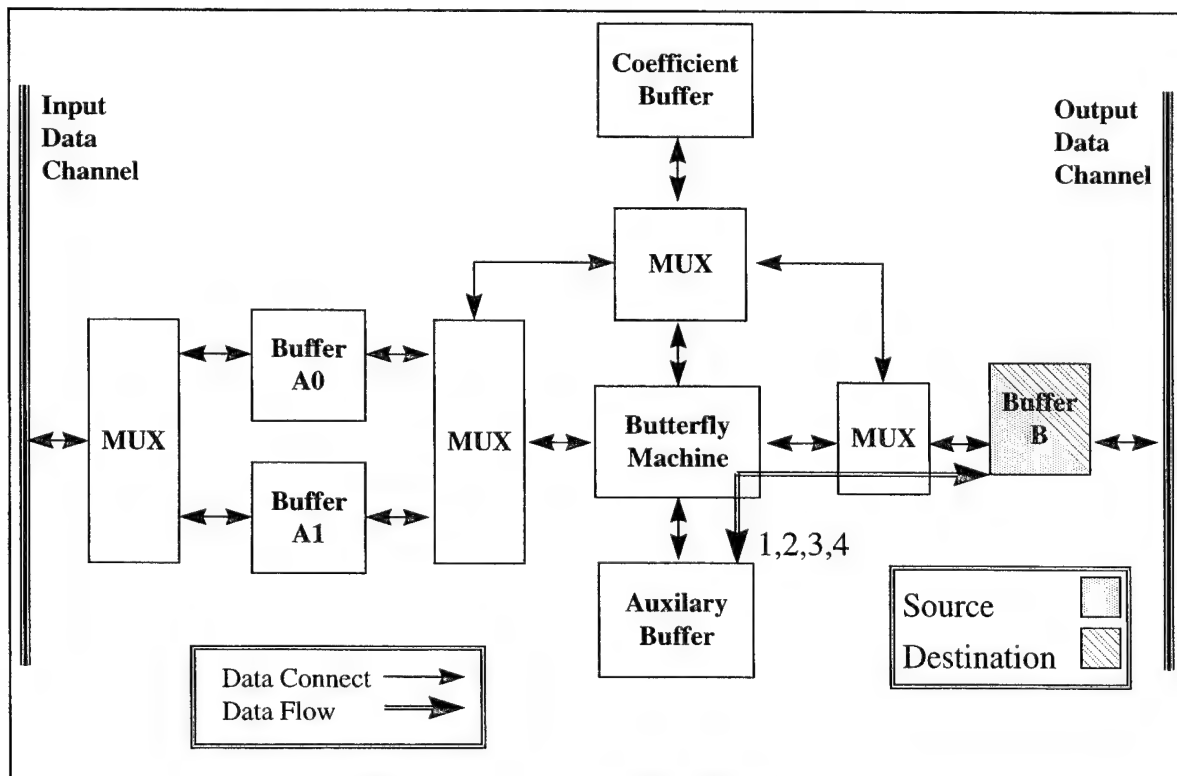


Figure III.21 SSCA Execution: N FFT



Observe that it is possible to choose appropriate paths such that the final result is in buffer B. If the number of passes were odd, one pass would be made to buffer A0 (e.g., for three passes, from buffer B to buffer A0, to the auxiliary buffer, and finally back to buffer B).

A super block is used for each of these basic blocks in order to repeat each block the appropriate number of times.

The number of cycles necessary to compute the SSCA, without taking into account latency, using a BFM is

$$C_{BFM} = \frac{N'N}{2} + 8N + 4N \log_{16} N' + \frac{N'N}{2} \log_{16} N. \quad (\text{III.30})$$

The number of cycles necessary to compute the SSCA with a conventional processor that is fully pipelined (i.e., each addition and multiplication can be accomplished in a single cycle) is

$$C_{GPP} = N(N' + 4) + (12N) \log_2 N' + \frac{3NN'}{2} \log_2 N. \quad (\text{III.31})$$

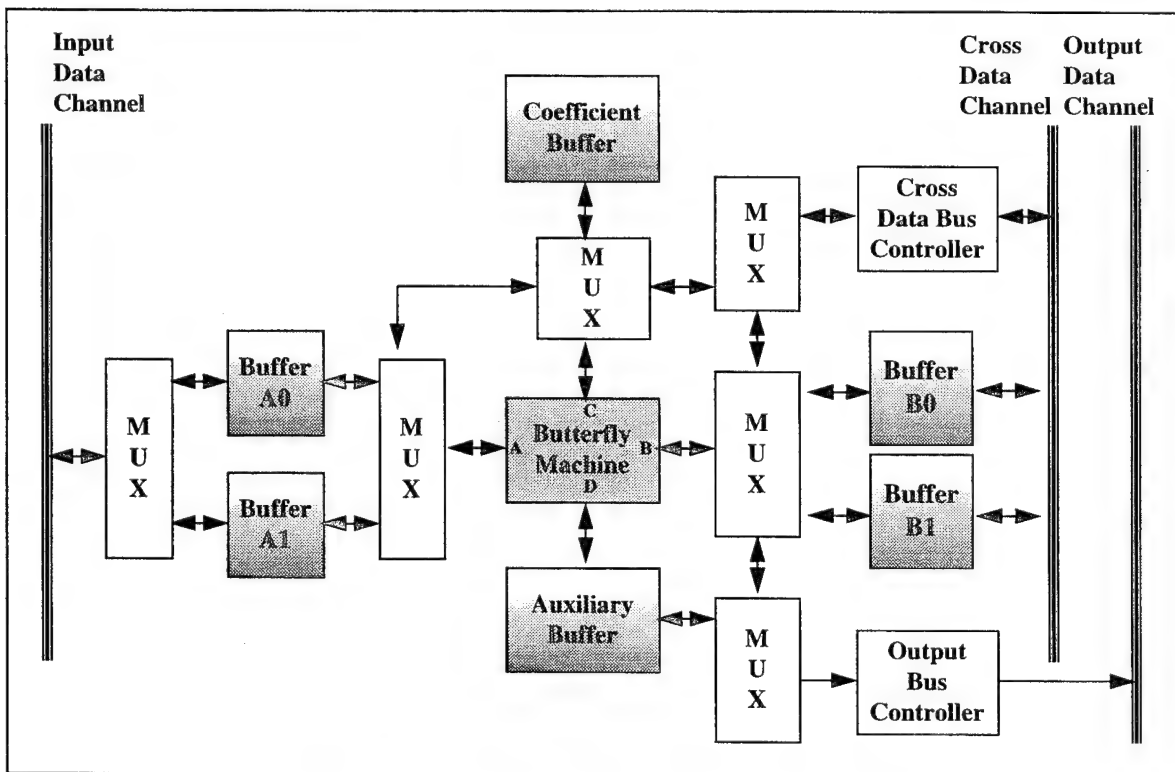
Computation of the total number of cycles for BFM and a fully pipelined general-purpose processor for  $N'$  varying between  $2^4$  and  $2^{10}$  and  $N$  varying between  $2^{16}$  and  $2^{22}$  reveals that there is approximately a 18 to 1 processing gain obtained with the BFM, relative to the fully pipelined general-purpose processor. Note that the expression for a fully pipelined processor is a theoretical upper bound. The ratio for an actual general-purpose processor would be much higher. This factor reflects the parallelism inherent in the butterfly processor.

## G. PARALLEL ARCHITECTURE

One board of the *parallel architecture* is shown in Figure III.22. The parallel architecture consists of two or more of these boards with a common input data channel, cross data channel and output data channel. A single board of this architecture is similar to that of a one-chip architecture with the addition of the cross data bus for inter-BFM communications. This parallel architecture represents a tradeoff between programming

flexibility and performance. A BFM architecture that is optimized for the SSCA is described in Loomis [Ref 41].

Each processor has an independent clock for each bus and processor. Data communication is accomplished with a message passing scheme. A message consists of a control packet consisting of a message id, message type, data packet length, the number of additional parameters, and the additional parameters. When a processor is ready to send data to a processor, it first sends the control packet. If the message is accepted, a ready to receive signal is passed back and the data transfer begins. The two types of transfers possible on the bus are "one to one" and "one to many".

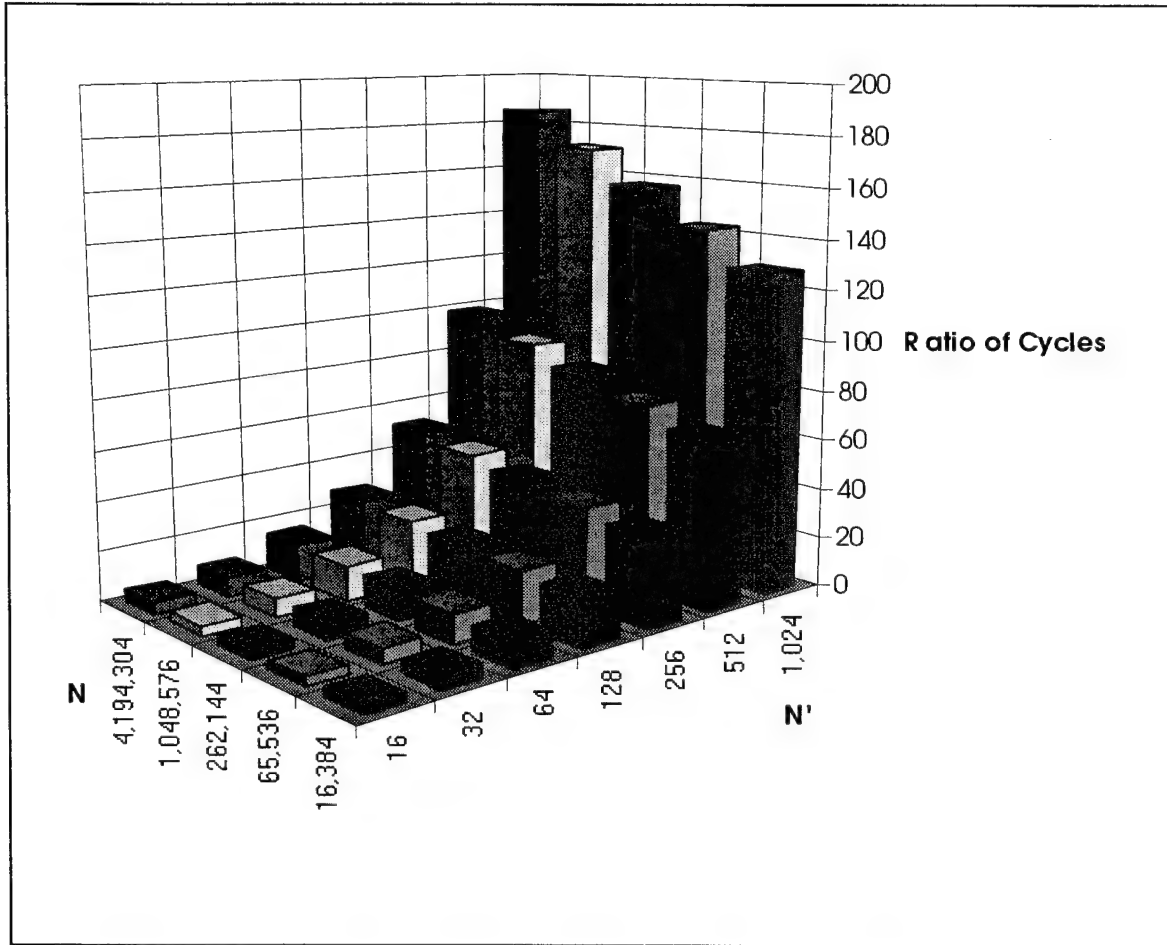


**Figure III.22 Parallel Architecture (One Board)**

The relative number of computations required for channelization versus the correlation multiplies and  $N$  FFTs varies considerably with the input parameters  $N'$  and  $N$ . The ratio of the number of cycles required for the back-end (i.e., the correlation multiply and the  $N$  FFT) versus channelization is

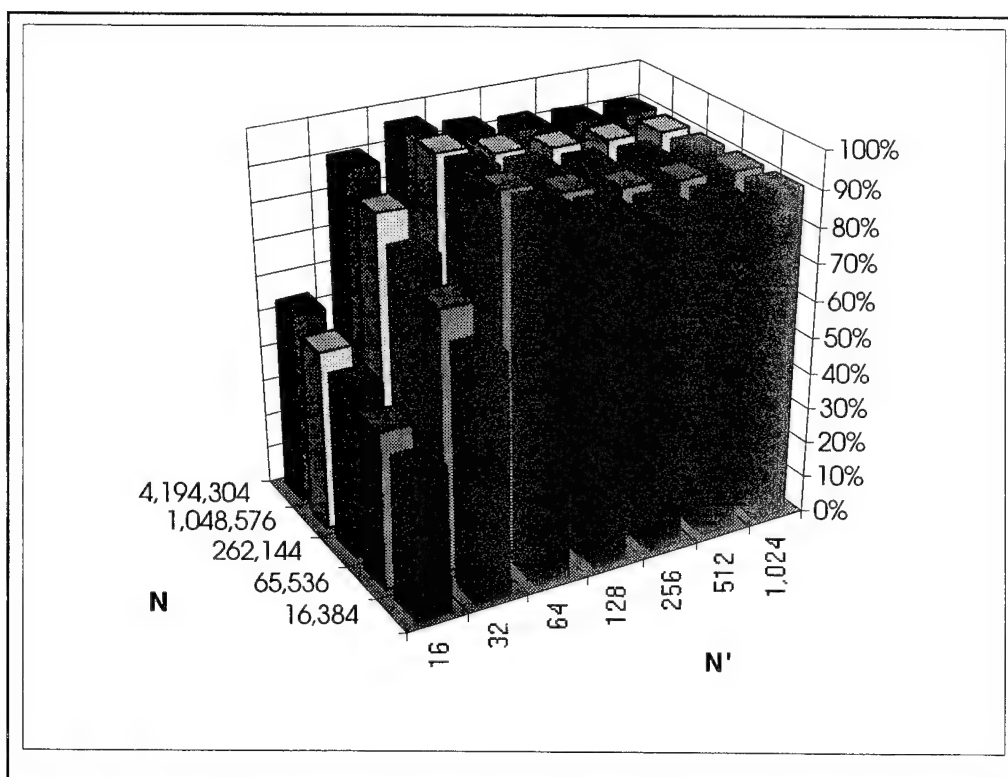
$$R_{BE-Ch} = \frac{NN' \left(1 + \lceil \log_{16}(N) \rceil\right)}{8N + 4N \lceil \log_{16}(N') \rceil} \quad (\text{III.32})$$

and is shown in Figure III.23 for various  $N'$  and  $N$ . Processors are allocated statically based on this ratio.



**Figure III.23 Process Allocation: Ratio of Backend to Channelizer Cycles**

Execution of SSCA using the parallel architecture is similar to that of the one-chip architecture except that blocks must be allocated to processors. The simplest scheme based on the data shown in Figure III.23 is to dedicate one BFM to channelization and the remainder to the back-end processing. The efficiency obtained using this approach is illustrated for a ten processor system in Figure III.24.



**Figure III.24 Processing Efficiency for SSCA (Ten Processor System)**

## IV. DESCRIPTION OF SPLIT TRANSACTION MEMORY

### A. PHYSICAL DESCRIPTION

Split Transaction Memory (STM) is a memory architecture that is designed to support a vector processing architecture with a throughput that approaches one, as defined in Equation (II.4). STM takes advantage of addressing patterns that are characteristic of constant stride. Of particular importance is the ability of STM to support radix- $r$  and digit reversal address patterns where  $r$  is a power of two.

STM provides better throughput than cached memory because it takes advantage of the predominate characteristic found in the butterfly machine architecture: patterns of constant stride and particularly constant strides of powers of two. Although the memory reference patterns exhibit some locality of reference, the data sets are frequently too large to support a caching strategy.

STM is an implementation of standard interleaved memory that takes advantage of more than one local buffer in each bank. To customize the memory system to the target problem domain, (e.g., a vector architecture supporting cyclostationary processing), STM incorporates a memory decoding scheme based on permutation decoding. In particular, this version of STM is designed to provide a throughput that approaches one for radix- $r$  and digit reversal address patterns as well as address patterns of constant stride.

STM is based on the premise that some latency can be absorbed by the processor. In particular, memory requests can be made in advance of completing the current instruction. In general, memory requests may be made so long as the memory system has the capacity to accept the request. Memory capacity will be described later in this section.

A high-level view of this concept is shown in Figure IV.1. Memory is partitioned into  $k$  banks. Each bank consists of a *smart cache* and a bulk storage module. The smart cache contains memory referred to as *cache elements* that operates at the same speed as the processor. The BSM-CE controller and CE-bus controller are responsible for interfacing the cache elements with the bulk storage and the system data/address buses respectively. The CE-bus controller drives two control lines that are used for handshaking with the processor.

One line is used to signal when memory requests can be made by the processor. The other line is used to indicate when memory responses are available to the processor.

When the processor makes a memory access, the bank which recognizes the memory access latches the request (i.e., the address, whether it is a read or write request, and data if it is a write request) into a cache element. A cache element (CE) is that set of data necessary to support one memory access to the memory bank. The cache element's *in-use* bit, is also set. When the cache element has been processed (i.e., data has been either written to or read from the bulk storage for a write or read access respectively), the cache element's *ready* bit is set.

The components of a cache element are illustrated in Figure IV.2. Each request is uniquely identified with an index which is used for synchronization of memory accesses with the processor. This synchronization will be discussed in the context of the request and response counters later in this section.

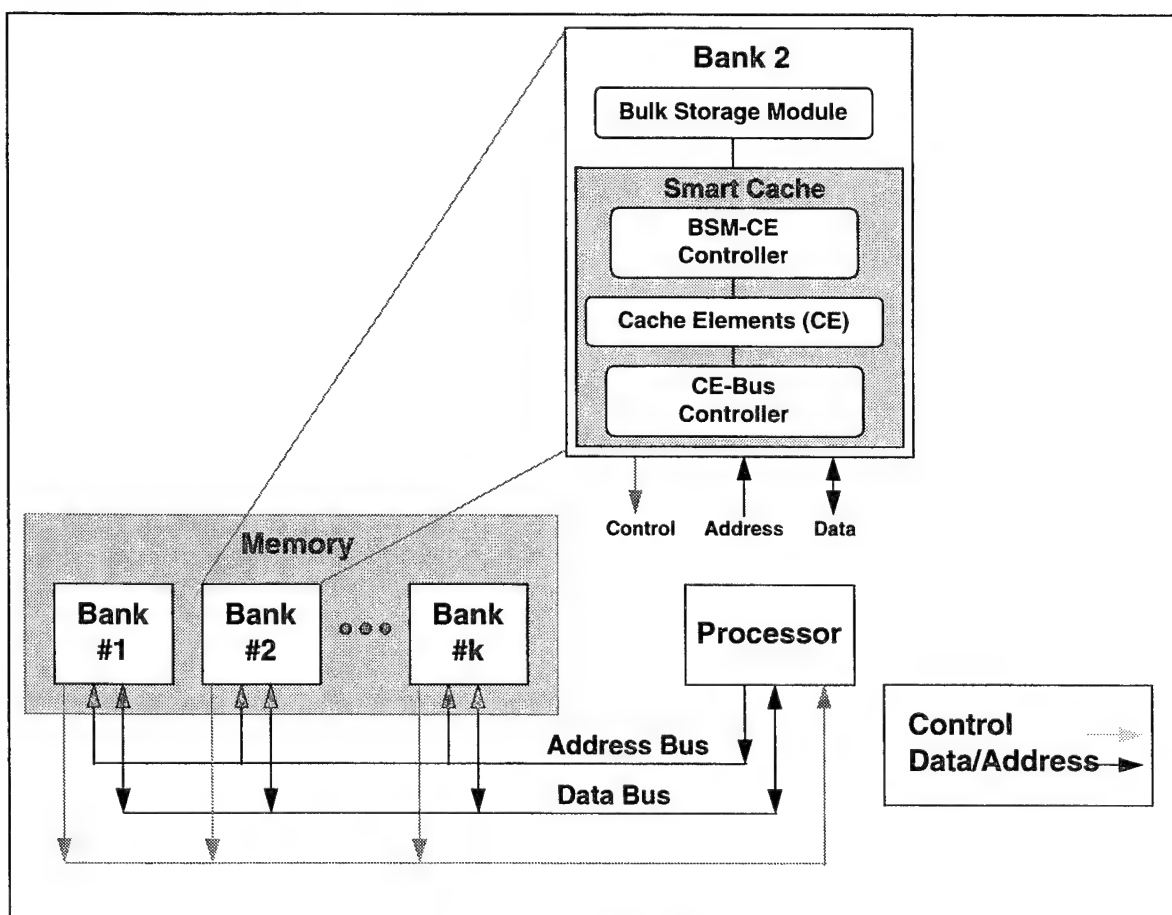


Figure IV.1 Split Transaction Memory Overview

The smart cache manages cache elements and requests for memory read and write accesses. There are three activities that take place on each clock cycle:

- Memory requests are recognized and accepted if the memory system has available capacity. The concept of memory capacity will be explored later in this section.
- Memory read responses, ready for the processor, are placed on the data bus in a synchronized order.
- The BSM, when not busy, is tasked with the next pending read or write operation.

Request Index	Address	Data	In Use bit	Read/Write bit	Ready bit
<b>Request Index</b>	<b>Value obtained from Request Counter.</b>				
<b>Address</b>	<b>Physical address.</b>				
<b>Data</b>	<b>Value to be read or written.</b>				
<b>In Use Bit</b>	<b>Bit indicating whether cache element is available for use.</b>				
<b>Read/Write Bit</b>	<b>Bit indicating the type of memory access type..</b>				
<b>Ready Bit</b>	<b>Bit indicating whether memory request has been serviced. This is used for a read transaction to indicate that the data has been retrieved from DRAM.</b>				

**Figure IV.2 Cache Element**

Memory accesses are usually initiated within a bank without waiting for previous access responses to be completed, either within the bank or from the memory system in general. For a read access, the smart cache retrieves the required data from the bulk or main storage and stores it into the associated cache element of the smart cache. A write request is sent by the processor to the smart cache. This data is later written to the bulk storage by the BSM-CE controller. For the design that follows, the BSM-CE controller processes requests in the order they were requested within a bank. All memory accesses are returned to the processor in the order that they were requested (for all banks).

Coordination of the STM is accomplished with two counters and two control lines. The two counters, *request counter* and the *response counter*, are used to link a memory request and the associated response for read accesses. These counters are shown in Figure IV.3. Initially, the request counter and the response counter are set to zero. When a memory request is accepted from the processor, the value of the request counter is placed into the *request index field* of the cache element (see Figure IV.2) that is used to store the request. The request counter is then incremented by one.

The response counter contains the index for the next read response needed by the processor. When a CE-Bus controller detects that a read response is ready for a cache element and its request index is equal to the current value of the response counter, a memory response cycle is performed. The CE-Bus controller associated with this response, places the associated data contained in the cache element onto the data bus. The response counter is then incremented by one. The latency of a memory access is

$$L = (\text{Request Counter} - \text{Response Counter})T_{bus} \quad (\text{IV.1})$$

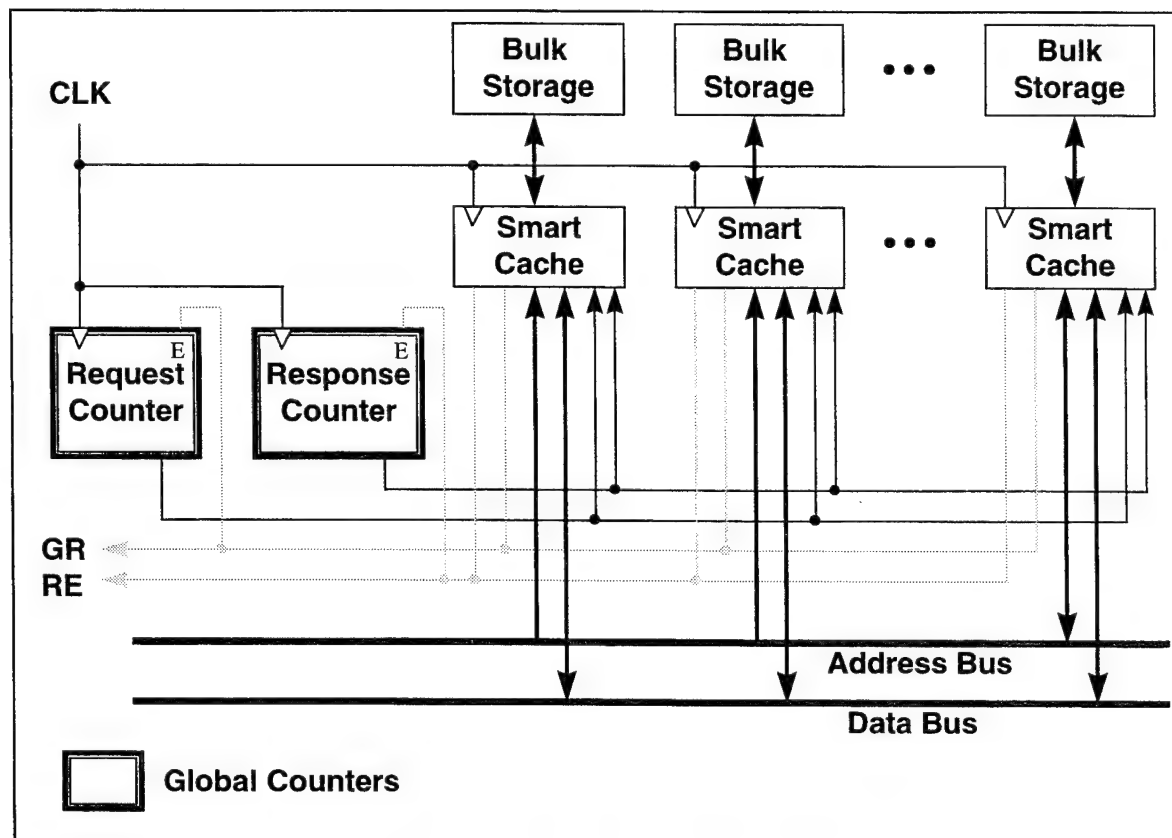
where  $T_{bus}$  is the bus cycle interval. This expression does not take into account for latency as a result of a stall.

A key issue of the STM design is the mapping of addresses to bank numbers and indices within a bank. Several methods were described in Chapter II Section D that can be used. Conventional interleaving results in poor performance for FFT related memory reference patterns when the radix of the FFT and the number of banks are both powers of two. One solution is to pick a stride and number of banks that are relatively prime. Two strategies described in Chapter II select a prime number of banks. These solutions either incur excessive propagation delay in the bank selection hardware, or assume an addressing pattern that is not appropriate for the butterfly machine architecture.

The two control lines used in this design are the *grant request* and *response enable* control lines. Memory access requests by the processor are controlled by the grant request (GR) control line. Each memory bank's CE-Bus Controller enables the GR as long as there are available cache elements. All GR lines from the banks are wire-ORed to form a single output signal to the processor resulting in a single ready signal for all of the memory banks.



This is done to provide a simpler interface to the processor, rather than having a line for each memory bank. If any bank does not have cache elements available (i.e., if it is “full”), the GR line becomes inactive and the processor will refrain from further memory accesses until a cache element in the full memory bank is freed.



**Figure IV.3 Top Level Memory System**

The response enable (RE) control line performs a similar role to that of GR, but for memory responses. Each bank's CE-bus controller generates a RE signal that is in turn wire-ORed to form a single control line to the processor. The default for the RE line is to be disabled. The next response required by the processor (i.e., the one pointed to by the response counter), can only be serviced by one bank. If the response is for a read and the data has been retrieved from bulk storage, the RE line is enabled by that bank and the data is placed on the data bus for the processor. The response counter is then incremented.

One design of the STM smart cache is shown in Figure IV.4. The data and address buses and the read-write line enter the smart cache in the upper left-hand corner of Figure IV.4 labeled Data, ADDR, and R/W respectively. The cache elements are located in the

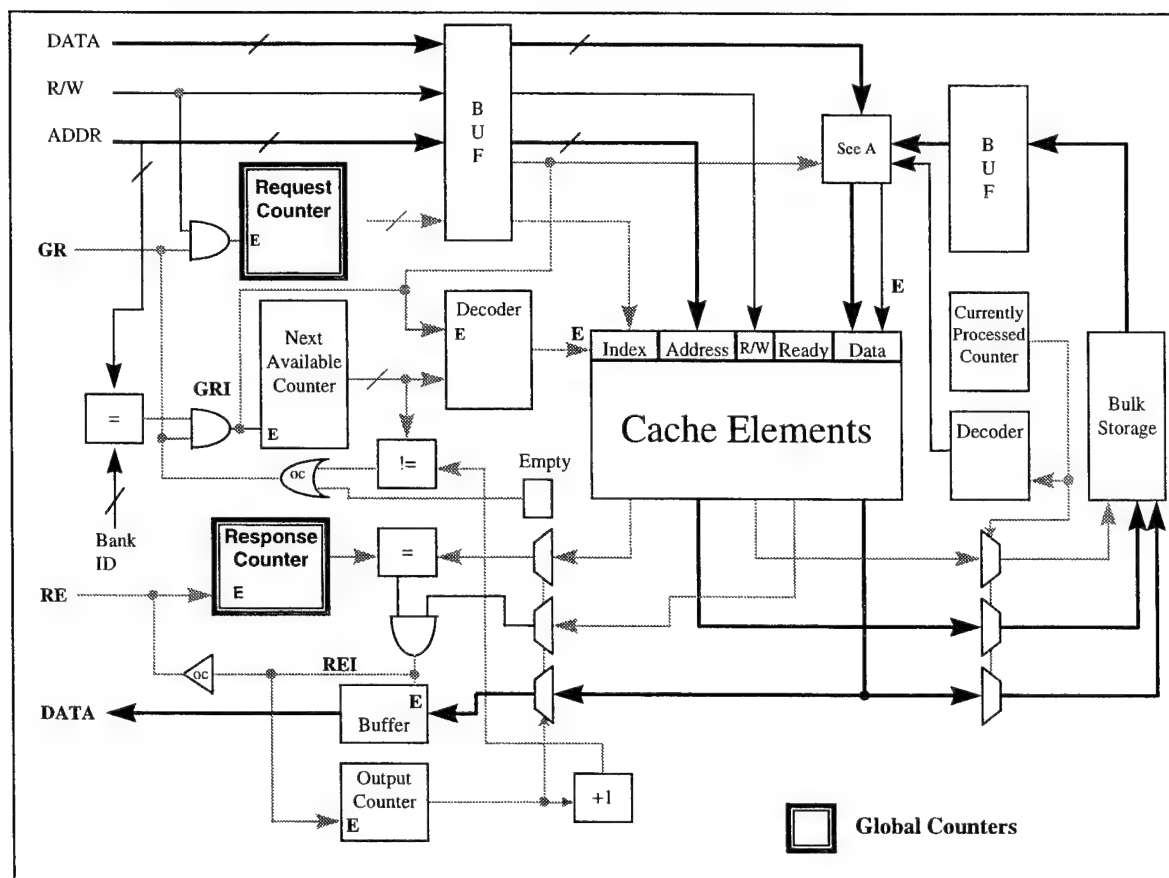
middle of the figure with the CE-Bus and BSM-CE controllers located to the left and right of the cache elements. The bulk storage module may be found in the far right of the figure.

Notice that in this design, the request and response counters are logically global (i.e., at any instant in time, there is a single value for each counter). However, the value of the counters are maintained on each smart cache as a hardware counter and the global signals GR and RE are used as a control signal to increment all request counters and response counters respectively in a memory system rather than a single request counter and response counter as shown in Figure IV.3.

Before looking at the specifics of this design, it is useful to describe the semantics of three counters in the smart controller. The definitions for three counters follow:

- Next Available Counter (NAC) - This counter is used as an index to the next available cache element to be used to store a new memory request.
- Currently Processed Counter (CPC) - This counter points to the next cache element that has a memory request pending for the bulk storage module that has not been completed.
- Output Counter (OC) - The output counter points to the cache element that will contain the next memory read response in the bank.

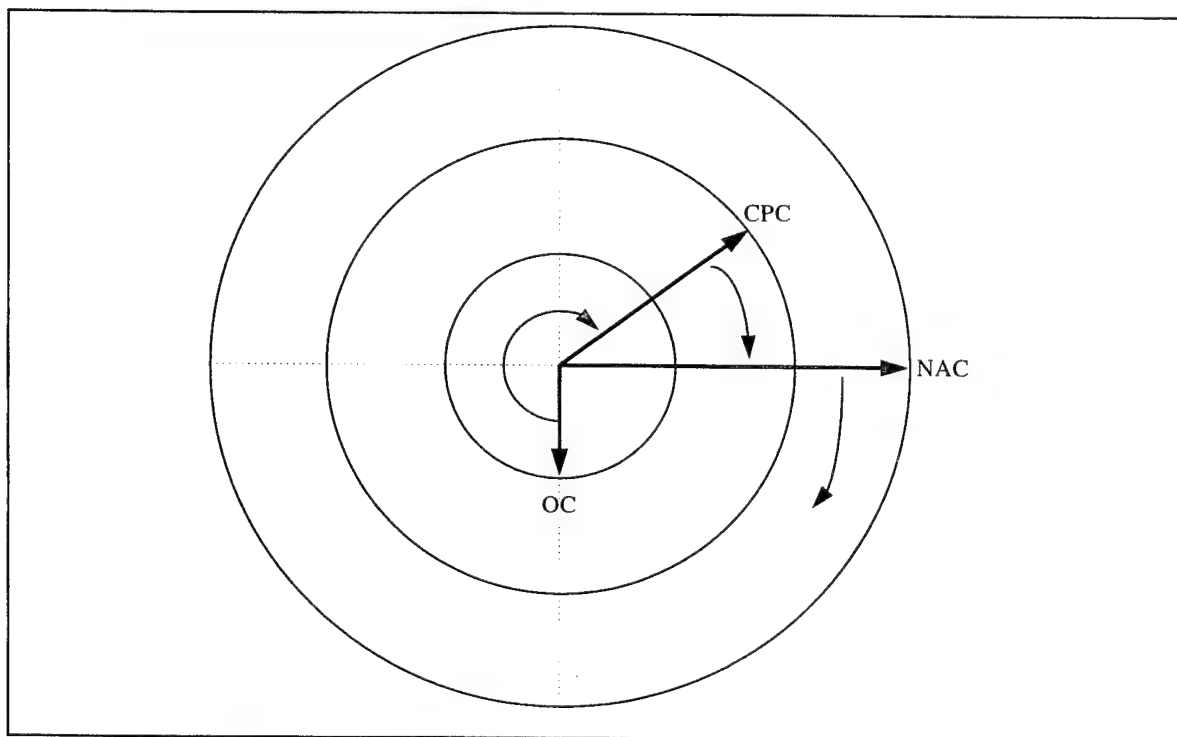
The relationship among the three counters is illustrated in Figure IV.5. All three counters are initialized to zero. This is the defined condition for an empty memory bank. By empty, it is meant that the bank has no pending memory requests. The NAC always points to the next available free cache element. As memory requests are accepted by the bank, the NAC is advanced after each request accepted. The CPC follows the NAC and advances whenever a memory request to bulk storage is completed by the BSM-CE controller. If the bank is kept busy (i.e., if there is always a memory request to process by the BSM-CE controller), this counter will generally advance at a frequency of the memory ratio. Finally, the OC advances whenever a read request is processed and passed back to the processor. The OC is also advanced whenever it points to a cache element containing a write request.



### Figure IV.4 Smart Cache Design

The relationship of these three counters can be thought of as pointers to a circular queue where the NAC is the lead pointer with the CPC following the NAC, and the OC following the CPC. Several possible conditions may arise and can be illustrated with Figure IV.5. If all three counters point to the same location, the memory bank is empty (i.e., no pending memory requests for the bulk storage or output responses). This is the initial state of the memory bank. If  $NAC == CPC$ , then there are no pending memory requests for the bulk storage for the bank. If  $CPC == OC$ , then there are no pending read responses from the bank. The three counters can be thought of as chasing one another (CPC chasing the NAC, OC chasing the CPC, and finally the NAC chasing the OC.) Observe that the CPC is allowed to catch up with the NAC and the OC is allowed to catch up CPC. However, NAC is not allowed to catch OC. If  $(NAC+1) == OC$ , then no cache elements are available and the memory is said to be full. This definition for an available cache element utilizes  $k-1$  of the cache elements rather than all  $k$  of them. This is done to simplify the logic for detecting when the memory bank is empty or full.

Returning to Figure IV.4, the CE-bus controller may be divided into two principal components: that part that is responsible for accepting memory requests and the other that is responsible for coordinating the read memory response. Accepting memory requests is accomplished with the request counter, the next available counter, and the logic required to drive the GR signal. As indicated above, the request counter and response counter are initialized to zero. The request counter serves as the input to the request index register for the selected cache element. The NAC is a modulo- $k$  counter where  $k$  is the total number of cache elements in a memory bank. The NAC is used in conjunction with the decoder, to select the cache element to be used for the next memory request. The NAC and decoder are enabled with the GR Internal (GRI) signal, resulting in the increment of the NAC and the selection and loading of the cache element registers. The request counter increments whenever any bank accepts a memory read request. The request counter is enabled with GR.



**Figure IV.5 Relationship Between Smart Cache Counters**

The logic within a bank, driving the GR line, is the logical ORing of two conditions. This condition is:

$$((OC + 1) \sim = NAC) \text{ OR Empty.} \quad (IV.2)$$

The first condition tests whether there is an available cache element as described above. The second condition, whether the bank is empty, is specified with the empty flag. The GR internal (GRI) line is the logical AND of the GR line and the logic determining whether a bank is selected. The quantity labeled Bank ID is compared to  $n$  address lines to make this determination. The least significant address bits are assumed unless otherwise indicated in this study. The Bank ID may be stored in a register set by either with hardware switches or software.

The output response is implemented with the response counter, output counter, and the logic required to drive the RE and RE internal (REI) signals. The response counter is incremented whenever RE is active. RE is the wired logical ORing of each bank's REI line. For each bank, REI is the logical ANDing of two conditions,

$$(\text{Response Counter} = \text{Index}[OC]) \text{ AND Ready}[OC]. \quad (IV.3)$$

The first part of the condition checks whether this bank contains the next memory read response to be sent to the processor. The second condition is a check to ensure that the data has been acquired from the bulk storage. Data[OC] is passed to the data bus by enabling the tristate buffer.

The BSM-CE controller is responsible for managing requests to the bulk storage. The currently processed counter is a modulo- $k$  counter pointing to the cache element to be processed by the bulk storage. The CPC selects the appropriate cache element to be processed using a multiplexer. For read requests, the resulting data is written into the appropriate cache element through the use of a decoder.

The cache elements obtain data from either the data bus for memory writes or the BSM for memory reads. The logic in Figure IV.6 illustrates the interaction between the data sources, control lines, and registers for one cache element.

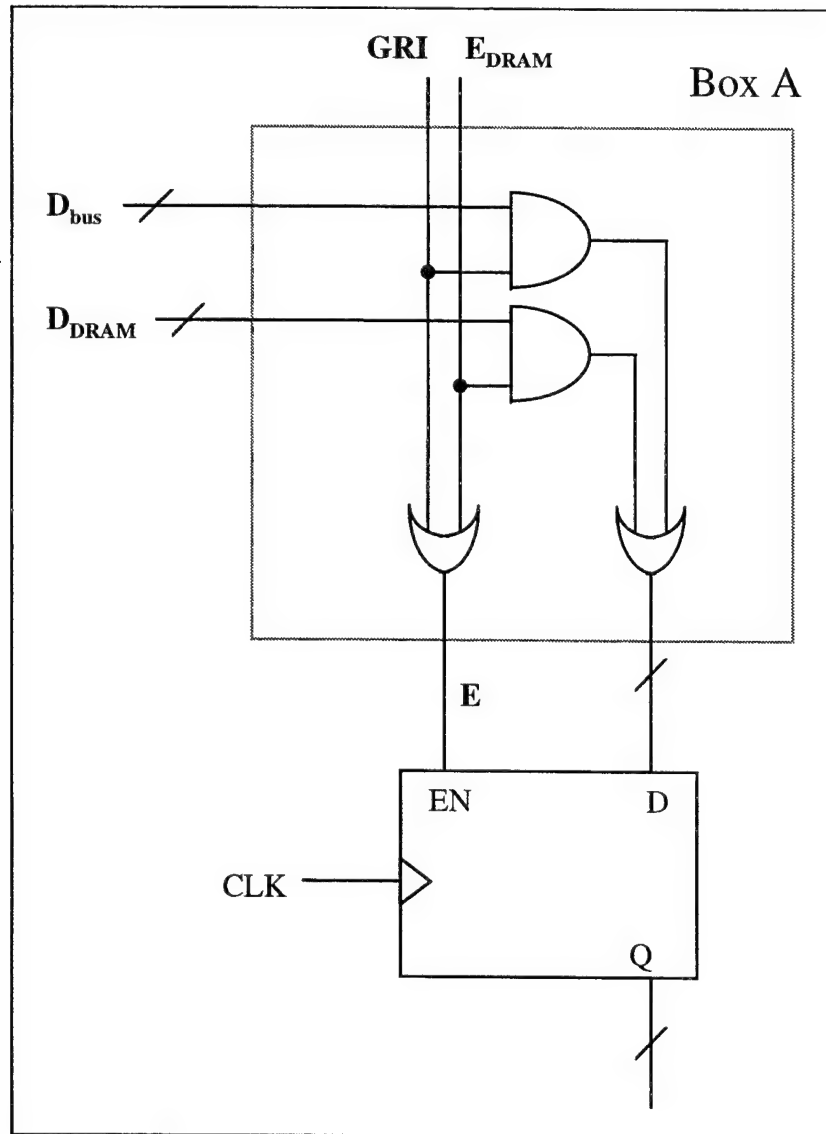


Figure IV.6 Block A for Figure V-4

## B. SIMULATION MODEL

The *STM simulation program* was written to explore characteristics of STM memory designs and to determine their effectiveness for a given memory reference pattern. The relationship and interaction between the STM simulation and related computer programs is shown in Figure IV.7. The STM simulation program is referred to simply as STM in the figure. All programs were written in Matlab<sup>TM</sup> and can be found in Appendix A.. The following discussion of the STM simulator will be partitioned into the *signal generators*, STM simulator, and the *graphics programs*.

### 1. Signal Generators

The STM simulator accepts three parameters that define the memory system and the memory reference stream that the STM simulator will be given to process. The memory reference stream is contained in a file referred to in Figure IV.7 as the *address stream*. The address stream is a list of integer pairs, the first representing an address of the reference and the second a flag indicating whether it is a read or a write operation.

Four programs (*gen\_const*, *gen\_cfft*, *gen\_dr*, and *gen\_rand*), referred to collectively as signal generators in the figure, were written to generate different classes of memory reference streams to be used as inputs to the simulator. The first three programs, *gen\_const*, *gen\_cfft* and *gen\_dr* were written to generate address patterns common to digital signal processing applications. The *gen\_rand* program provides an address stream that yields a random address pattern.

The first program *gen\_const* generates the most basic address pattern for vector processors; patterns of constant stride. The program *gen\_const* interface is

`ResultVect = gen_const(N, Stride, fname)`

where:

N - Number of addresses to generate

Stride - Stride of the pattern, and

fname - Name of the file containing the resulting address patterns.

The program **gen\_cfft** generates memory reference patterns consistent with constant geometry fast Fourier transforms (FFT) with butterfly passes with a radix of **R**. The program **gen\_cfft** has the following calling interface:

**ResultVect = gen\_cfft(N, R, fname)**

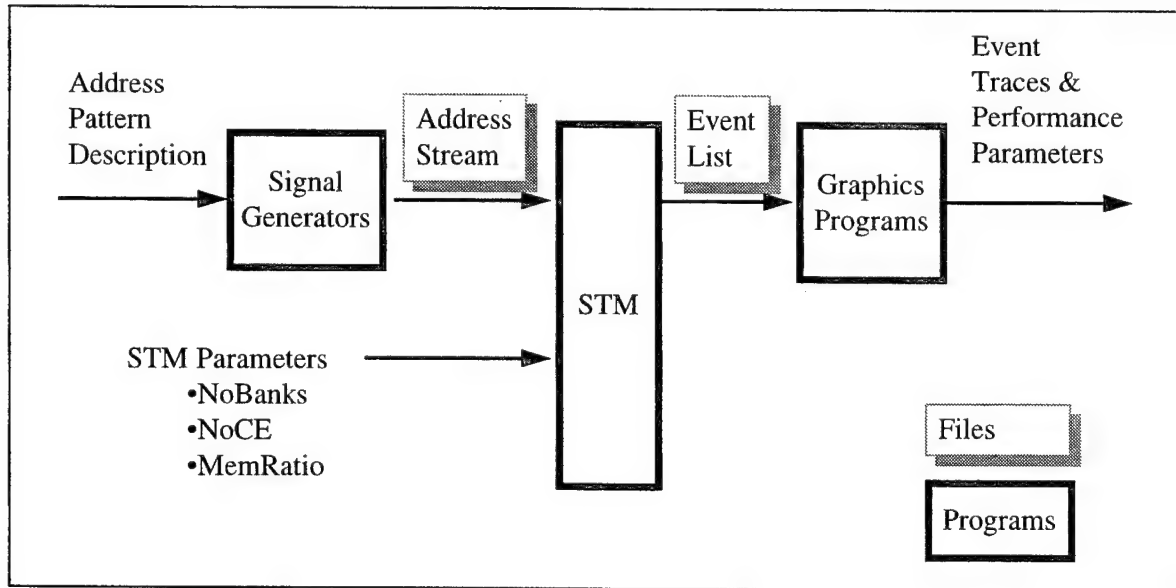
where:

**N** is the number of addresses to generate,

**R** is one of the factors of **N**, such that  $N = N1 * R$ , and

**fname** is the name of the file containing the resulting address patterns.

For realistic patterns, it is expected that  $R \ll N1$ .



**Figure IV.7 STM Simulation Overview**

The program **gen\_dr** provides for generating digit reversal patterns necessary for one pass of an FFT. The digit reversal pass is found in the last pass of a FFT for the class of FFT algorithms described in Section D of Chapter 0.

Table IV.1 illustrates the bit reversal pattern for a base of two with three digits. The program **gen\_dr** has the following calling interface:

**ResultVect = gen\_dr(NoDigits, Base, fname)**



where:

NoDigits is the number of digits in the address pattern,

Base is the base of the number system used, and

fname is the name of the file to store the resulting address patterns.

Normal Pattern (Base 10)	Normal Pattern (Base 2)	Bit Reversed Pattern (Base 10)	Bit Reversed Pattern (Base 2)
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

**Table IV.1 Digit Reversal for Three Digits Base 2**

The last program, **gen\_rand** generates a sequence such that the probability that the next address will be sequential or linear is  $p$ , and therefore a probability of  $1-p$  that the next address will be a jump to a random address. The calling sequence for **gen\_rand** is:

**ResultVect = gen\_rand(N, p, seed, fname)**

where:

**N** is the number of addresses to generate,

**p** is the probability that the next instruction is the next address,

**seed** is the random number generation seed, and

**NoBanks** is the number of banks in simulation.

This function is used to generate address patterns characteristic of general-purpose computing. When  $p > 0$ , there exists a sequential address characteristic that simulates address patterns that occur when fetching instructions. The random nature of the simulated address pattern captures data references as well as program branching.

## 2. STM Simulator

The STM Simulation program consists of the function `stm` and a collection of support functions called by `stm`. The program `stm` accepts a memory address stream described above, and three parameters that define the memory system:

- number of memory banks (`NoBanks`),
- the number of cache elements per memory bank (`NoCE`) and,
- the ratio between the dynamic memory cycle time to the static memory cycle time (`MemRatio`). This parameter specifies the number of static memory cycles required to complete one dynamic memory cycle.

The calling sequence for `stm` is:

`stm(Fname,ASCII,Level,AList,NoBanks,NoCE,MemRatio,MemDecode,A)`

where

`Fname` is the file name of the saved data,

`ASCII` specifies the format of the output file (either ASCII or binary).

`Level` specifies the level of detail of output saved in `sf_name`. There are three levels of detail that can be saved. Level 0 is a complete dump of all of the memory bank registers for each clock cycle. This level is used primarily for test and validation of the program. Level 1 provides a tabular listing of events. Level 2 generates a file suitable for input into the graphics programs.

`AList` is the memory address list. This is a matrix where each row is of the form:

[Address RW Flag]

`NoBanks` specifies the number of banks to be used in the simulation,

`NoCE` is the number of cache elements to be used in each bank of the simulation,

`MemRatio` is the ratio of dynamic memory cycle time to static memory cycle time,

`MemDecode` is a flag that specifies the type of memory decoding, and

A is the permutation matrix when MemDecode=1 and undefined otherwise.

The program `stm` models all of the counters, registers, and flags for each bank of the memory under simulation. These variables can be categorized as bank counters, global counters, the cache element array, control signals, flags, and other variables. The following three variables are used to model the counters for each memory bank:

- **Next Available Counter (NAC)** This counter points to the next available cache element (CE) available for a memory (read or write) request.
- **Output Counter (OC)** This counter points to the next CE containing a read that has data ready to be sent back to the processor.
- **Currently Processed Counter (CPC)** This counter points to the CE that is currently involved in either a dynamic read or write cycle when `PDC == TRUE`.

The program uses global counters as shown in Figure IV.3, rather than replicating them in each memory bank as indicated in Figure IV.4. They are defined as:

- **Request Counter (ReqC)** This counter is used to ensure that each read request is matched with the read response. `ReqC` is loaded into the next available CE's `ReqIndex` field during a memory request cycle. Note: This counter is conceptually global in that every memory bank has access to the `ReqC` contents.
- **Response Counter (ResC)** The response counter is also conceptually a global counter and is used in conjunction with the `ReqC`. `ResC` is compared with the `ReqIndex` value selected by output counter (OC). If they are equal, the corresponding `Ready` bit is set. `ResC` is incremented when a read response is returned on the bus.
- **Dynamic Memory Cycle Counter (DCount)** This counter is initialized to `ReqCount` at the beginning of a dynamic memory cycle and decremented for each system cycle.

The Cache Element (CE) of each bank is represented by the cache element array. The cache element array is the focus of the STM design. Each CE is a resource for processing one read or write request and is the central interface between the main system bus and the dynamic memory module. The cache element array consists of the following components:

- **Request Index Array (ReqIndex)** The index of the CE, addressed by the next available counter (NAC), is loaded with the value of the request counter (ReqC) when a memory reference is serviced. Note that this value serves as a unique identifier for sending data back to the requester (e.g., the processor).
- **Address Array (Address)** The Address of the CE, selected by the next available counter (NAC), is loaded with the value of the address bus (ADDR) when a memory reference is serviced.
- **R/W Bit Array (RW)** The RW bit of the CE, selected by the next available counter (NAC), is loaded with the value of the address bus signal indicating either a read or a write request when a memory reference is serviced.
- **Ready Bit Array (Ready)** The Ready bit of the CE, selected by the next available counter (NAC), is reset, indicating either data for a read request is not available or a write request has not been completed when a memory reference is serviced. This bit is set for a read request when the data has been loaded from the dynamic memory.
- **Data Array (Data)** The Data Array is used differently for read and write memory requests. For a memory read, the Data array of the CE, selected by the currently processed counter (CPC), is loaded at the end of a dynamic write cycle. This is followed by the Data array being read and passed to the Data Bus (DATA), when referenced by the output counter (OC) and the Request Enable (RE) line is active. For a memory write, the contents of the DATA bus is written into the Data array of the CE, selected by the next available counter (NAC).

The control signals for **stm** are defined as follows:

- **Grant Request Internal (GRI)** When active, this signal indicates that a bank is ready to accept requests from the processor.
- **Grant Request (GR)** When active, this signal indicates that the memory system is ready to accept requests from the processor. The **GR** signal is formed by a wired AND of all of the **GRI** signals.
- **Response Enable Internal (REI)** When active, this signal indicates that a bank is ready to send data requested with a read request, back to the processor.
- **Response Enable (RE)** When active, this signal indicates that the memory system is ready to send data requested with a read request, back to the processor. The **RE** signal is formed by a wired AND of all of the **REI** signals.
- **Bank Select (BS)** When active, this signal indicates that this bank has been selected for a memory access.
- **Start Dynamic Read Cycle (SDRC)** This signal indicates the beginning of a dynamic read cycle.
- **Start Dynamic Write Cycle (SDWC)** This signal indicates the beginning of a dynamic write cycle.

The following is a list of the flags defined in **stm**:

- **Empty** This flag is active whenever there are no memory requests in the smart cache.
- **Processing Dynamic Cycle (PDC)** This flag is active whenever the dynamic memory subsystem is processing a memory request.

**ReqCount** is a variable defined in **stm** that specifies the total number of system cycles required to process a dynamic memory cycle.

A simplified algorithmic description of **stm** is shown in Figure IV.8. The style used to describe the algorithm is borrowed from the Matlab language. Formal variables are not shown except where they provide clarity to the algorithm. Other details, such as file I/O, are also not shown.

The first function, **initialize()**, represents all one-time initialization required for **stm**, such as initializing counters and cache element arrays for the banks. The work is accomplished in the **WHILE** loop which executes until the variable **done** is set to **TRUE** by the function **simulation\_complete()**, which returns **TRUE** when there are no more addresses to process and when all of the memory banks are empty.

```
stm(AList,NoBanks,NoCE,MemRatio)
    initialize();
    done = FALSE;
    while ~done,
        GRI = evaluate_gri();
        REI = evaluate_rei();
        Empty = evaluate_empty();
        generate_address();
        for BankNo = 1:NoBanks,
            memory_response();
            service_dynamic_memory();
            service_memory_request();
        end;
        done = simulation_complete();
        save_results();
        System_Clock = System_Clock + 1;
    end;
```

**Figure IV.8 Simplified Algorithmic Description of **stm****

Each pass of the **WHILE** loop processes one clock cycle. Each memory bank is evaluated to determine the status of **GRI**, **REI**, and **Empty** with the calls to **evaluate\_gri()**, **evaluate\_rei()**, and **evaluate\_empty()**. The function

`generate_address()` then conditionally generates an address if the memory system is able to accept a new request. The conditional nature of the address generation is not shown explicitly here. Each bank is then evaluated in the FOR loop.

There are up to three events that may occur with each memory bank. These events are defined as follows:

- Accept a memory request,
- Generate a bulk storage memory cycle. This comes in three types, generate a read bulk storage memory cycle, generate a write bulk storage memory cycle, and generate a Processing bulk storage cycle.
- Send a read response.

These events are processed by the functions `memory_response()`, `service_dynamic_memory()`, and `service_memory_request()` respectively. The variable `done` is then set, results for this cycle are saved with the function `save_results()`, and the system clock is incremented.

The results are saved in a file that can be processed using graphics programs described in the next section.

### **3. Graphics Programs**

The graphics programs shown in Figure IV.7 provide graphic plots of the memory traces and compute scalar performance measurements of the simulation results. The primary graphics function is called `m_anal()`. This program produces plots that are used to obtain quantitative and qualitative insight into a particular simulation run. Its calling convention is as follows:

```
[ TP,S,MaxL,AvgL,StdL] =  
    m_anal(fname,ASCII,Apattern,WinLen,PlotFlag,Length,PrintFlag)
```

where

`fname` is the name of the file containing data produced by `stm` to process.

**ASCII** is a flag indicating whether **fname** is stored as ASCII or binary file. A 0 and 1 specifies binary and ASCII respectively.

**Apattern** is a short description of the Address pattern to be used for the title of the graph.

**WinLen** is the length of the smoothing window for computing instantaneous throughput.

**PlotFlag** is a flag indicating the number and types of plots to be produced. Valid values are 0 and 1, indicating no plots or one plot respectively.

**Length** specifies the number of points used in a plot. A value of 0 means use all of the points. Any value greater than 0 indicates the number of point to plot.

**PrintFlag** is a flag indicating the types of output desired. A value of 0 means print to the screen, a value of 1 means print to a postscript file, and a value of 2 means send directly to a printer.

This function produces the scalar output statistics of throughput (**TP**), speedup (**S**), maximum latency (**MaxL**), average latency (**AvgL**), and the standard deviation (**StdL**) for the simulation. These statistics may be an end result in themselves or they can be used as input into the graphics program **p\_mesh()** described below.

An example of the plot produced by **m\_anal()** is shown in Figure IV.9. The input parameters (i.e., a short description of the memory address stream, and the three parameters that define a STM memory) are shown above the top graph. The plot on the top is the instantaneous latency versus time. For this example, the latency begins at nine and becomes 16 at steady state.

Scalar performance parameters are displayed above the middle plot. These parameters are speedup, average throughput, maximum throughput, average throughput, and the standard deviation of the throughput. The middle plot is a moving average of the throughput based on a window of length **WinLen**. Those plots shown in this document were constructed with **WinLen** = 8 unless otherwise stated. The plot on the bottom is a time series display of the control lines grant request (**GR**) and request enable (**RE**). **GR** is



active if the line is high (at the GR level) and inactive otherwise. RE may be interpreted in a similar manner.

The second graphics function is called `p_mesh()`. This function is responsible for constructing a mesh plot of one type of scalar performance measurement (e.g., speedup) produced by `m_anal`. This type of plot is used to compare a set of performance measurements when two variables (`NoCE` and `NoBanks`) are varied over a range. In Figure IV.10, the performance variable speedup is plotted for `NoCE` ranging from 1 to 64 and `NoBanks` ranging from 4 to 64.

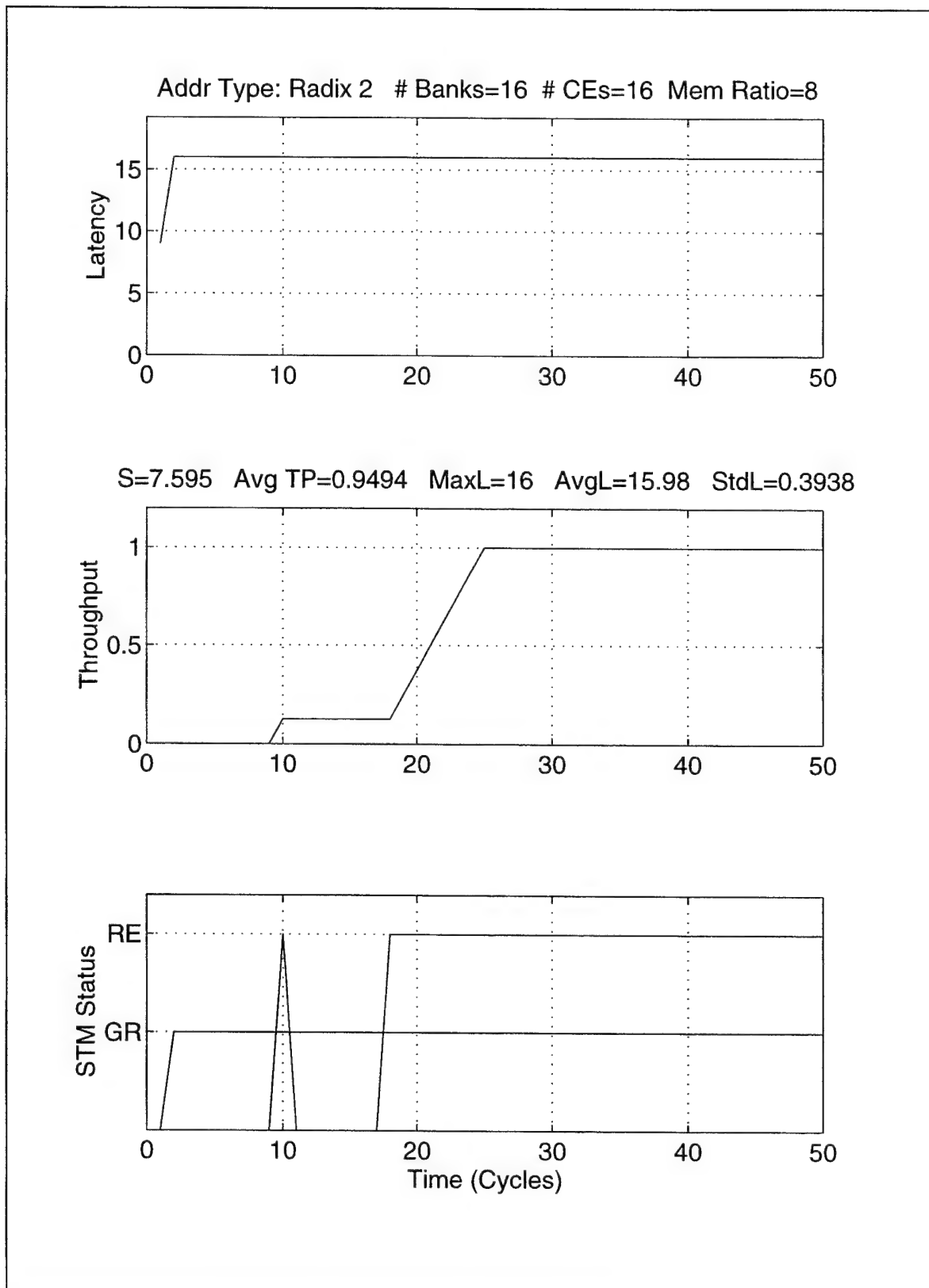
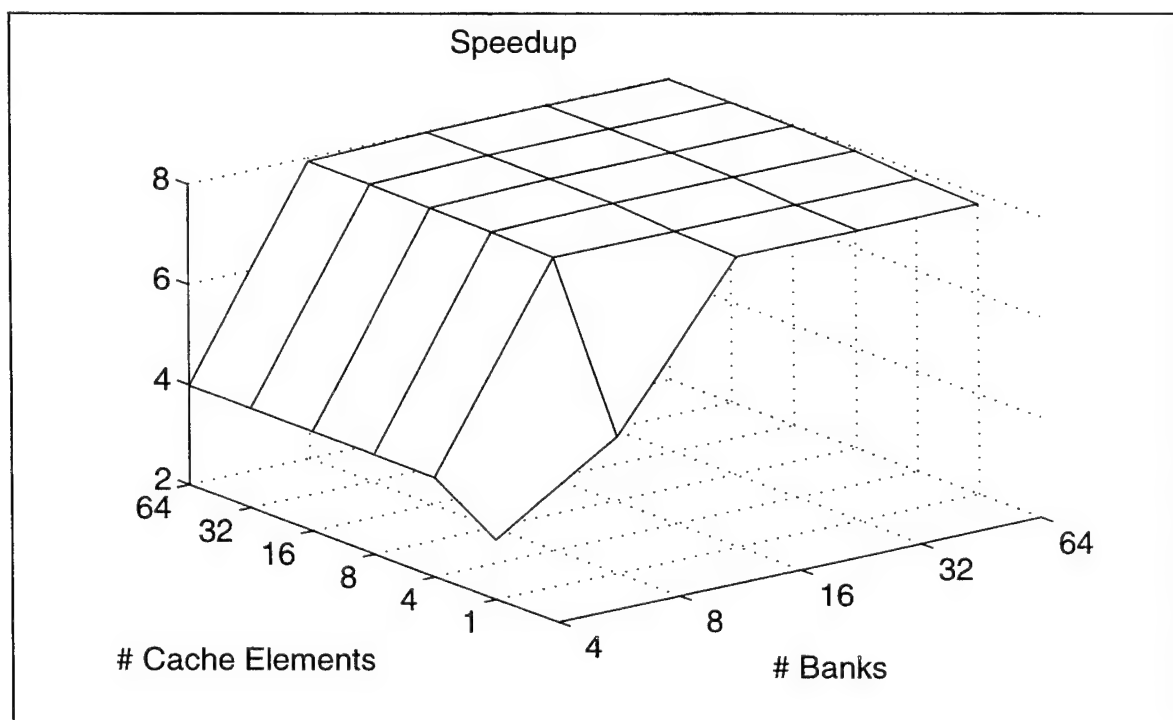


Figure IV.9 Example Plot From m\_anal Function



**Figure IV.10 Example Mesh Plot for the Performance Parameter Speedup**



## V. THEORETICAL PERFORMANCE ANALYSIS OF STM

In this section, performance of the STM will be investigated and described in terms of memory parameters and characteristics of the input memory reference vector. The conventional bank number decoding scheme is assumed as described in Section D of Chapter II for Sections A through C. In Section D, the effects of permutation-based decoding will be examined. The bank selection pattern will be described in terms of the characteristic of the input memory reference vector and the memory parameters. The bank selection patterns will then be used to determine expressions for the performance parameters, steady-state throughput ( $TP_{ss}$ ), and the maximum latency ( $L_{max}$ ).

The following analysis assumes that all memory references are read requests. This provides for the worst case analysis for STM. This analysis will begin with the simplest of the input reference streams, those streams with constant stride. Information gained from this analysis will then be used to address radix- $r$  butterfly and digit-reversed patterns.

### A. CONSTANT STRIDE

The parameters pertinent to performance measures for constant-stride address patterns are:

- Stride length ( $S$ ),
- Number of banks in the memory system ( $B$ ),
- Number of cache elements per memory bank ( $CE$ ),
- Ratio of bulk store to static memory cycle time ( $MR$ ).

The expression for the effective number of banks is repeated below for convenience. Given a stride  $S$ , and a number of banks  $B$ , the number of memory banks that will actually be used can be expressed as:

$$B_{eff} = \frac{B}{\gcd(S, B)}, \quad (V.1)$$

where  $\gcd(a,b)$  is the greatest common denominator of  $a$  and  $b$ .  $B_{eff}$  will be referred to as the effective number of banks. For example, if  $S$  and  $B$  are relatively prime, then  $B_{eff} = B$ . Alternatively if  $B$  is a factor of  $S$ , then  $B_{eff} = 1$ .

For the set of  $B_{eff}$  effective banks for a given input memory reference vector, the constant-stride address pattern distributes the addresses evenly with a size of one. By “evenly” it is meant that each of the effective banks is presented addresses in a round robin pattern. By “with a size of one,” it is meant that each bank is given one address at a time. Figure V.1 illustrates a memory system with  $B_{eff}$  banks, each bank with  $CE$  cache elements. This figure assumes that all of the banks are effective, or alternatively, only the effective banks are shown. The entries in each cache element represent the placement of the sequence number of each memory address where the addresses are distributed evenly with a size of one as described above. This is, in fact, the optimum placement within the effective banks because the work is spread evenly. At any point in time, the bank that will receive the next memory request will be the bank that is the least busy. Additionally, the use of input and output buffers for standard interleaving and cache elements for STM, provide pipelining of memory requests to each bank. This allows each bank to execute memory references with no wait cycles as long as there are memory references to process.

Bank #0		Bank #1		Bank # $B_{eff} - 1$	
CE Index	Addr #	CE Index	Addr #	CE Index	Addr #
0	0000	0	0001	0	$B_{eff} - 1$
1	$B_{eff}$	1	$B_{eff} + 1$	1	$2B_{eff} - 1$
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
CE-1	$(CE-1) B_{eff}$	CE-1	$(CE-1) B_{eff} + 1$	CE-1	$CE B_{eff} - 1$

**Figure V.1 Interleaved Memory Address Space: Conventional Bank Selection**

The round robin pattern coupled with the use of pipelining ensure that the bulk storage modules associated with the effective banks will be fully utilized for the constant-stride address pattern.

For any interleaved memory system, the steady-state throughput is determined by the memory ratio, the number of effective banks, and the efficiency with which the effective banks are utilized. As indicated above, there is full utilization for the constant stride pattern. The following discussion describes the relationship between the memory ratio and the number of effective banks.

The total number of bulk storage cycles ( $C_{BSM,r}$ ) required to process  $N_A$  memory requests is

$$C_{BSM,r} = N_A \cdot MR \quad (V.2)$$

The number of bulk storage cycles available ( $C_{BSM,\alpha}$ ) with a memory consisting of  $B$  effective banks ( $B_{eff}$ ) during  $N$  cycles is:

$$C_{BSM,\alpha} = B_{eff} \cdot N. \quad (V.3)$$

The banks can be assured to be used efficiently, for the reasons described above and therefore all of the available bulk storage cycles will be used. Setting Equation (V.2) equal to Equation (V.3), and applying the definition of throughput of Equation (II.6), yields:

$$TP_{ss} = \frac{C_{ideal}}{C_{actual}} = \frac{N_A}{N} = \frac{B_{eff}}{MR} \text{ for } B_{eff} < MR. \quad (V.4)$$

Note that the range of clock cycles used to compute the steady-state throughput is assumed to be in the steady-state region when applying Equation (II.4). The banks cannot process more memory requests than are available. If  $B_{eff} \geq MR$  then the maximum throughput is obtained, therefore:

$$TP_{ss} = 1.0 \text{ for } B_{eff} \geq MR. \quad (V.5)$$

If the number of effective banks is less than  $MR$ , then throughput will be proportional to the memory ratio as shown in Equation (V.4).

Under ideal conditions, latency is expressed as

$$L_{min} = MR + 2 \quad (V.6)$$

since  $MR$  cycles are required to process a memory request and one cycle is required for the input and another for the output of the memory request. If the number of banks is equal to or exceeds the memory ratio, the minimum latency is obtained for constant stride addressing patterns because there is sufficient memory capacity to process the memory requests, and the memory references are allocated efficiently to the banks. Therefore,

$$L_{max} = MR + 2 \text{ when } B_{eff} \geq MR \text{ and } NoCE \geq 2 \quad (V.7)$$

for STM and standard interleaving.

If the throughput is not optimum, (i.e.,  $MR > B_{eff}$ ), then the steady-state latency becomes a function of the memory ratio and the number of effective cache elements. Throughput not optimum implies that there are more memory requests than can be processed per unit of time. The steady-state latency associated with a constant-stride address pattern when the throughput is not optimal will be described shortly with the aid of Figure V.3.

The relationships between the performance measures and memory parameters for a constant-stride address pattern is illustrated in Figure V.2 and Figure V.3. The timing diagram in Figure V.2 is for a four bank memory (labeled B0 through B3) each with two cache elements indicated by the letters  $a$  and  $b$ . The top row is a clock for reference purposes. The row labeled *Bus* reflects the corresponding bank numbers of the address stream driven by the processor. The superscripts on the bank numbers are used to uniquely identify each memory reference.

The first memory reference,  $0^0$  is placed on the bus at clock cycle 0 and accepted by first cache element of bank 0 ( $B0a$ ) at clock cycle 1 as indicated by the entry  $0_{in}^0$ . The next four entries,  $p1$ ,  $p2$ ,  $p3$ , and  $p4$  indicate the time required for the bulk memory to process the memory request. The next entry  $0_{out}^0$  indicates that the memory response is passed back to the processor.

The memory ratio is four (as indicated by the  $p1$  through  $p4$  entries). Therefore, based on Equation (V.5), the memory will support maximum throughput for an address stream with constant stride of one which is suggested by the round robin bank pattern



shown on the bus. The first memory response occurs at clock cycle six with a latency of six. The memory system responds with an output every cycle thereafter yielding a throughput of 1.0. Note that in general, a memory request is accepted in a bank when the currently processed memory request is in its fourth cycle, thereby queuing up the new memory request just in time to keep the bulk memory continuously busy. By inspection, it can be seen that the latency is six for all memory references. There is substantial regularity in this example because of the constant stride of one address pattern and because there are sufficient banks to support a throughput of 1.0.

Clk	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Bus	0 <sup>0</sup>	1 <sup>0</sup>	2 <sup>0</sup>	3 <sup>0</sup>	0 <sup>1</sup>	1 <sup>1</sup>	2 <sup>1</sup>	3 <sup>1</sup>	0 <sup>2</sup>	1 <sup>2</sup>	2 <sup>2</sup>	3 <sup>2</sup>	0 <sup>3</sup>	1 <sup>3</sup>	2 <sup>3</sup>	3 <sup>3</sup>	0 <sup>4</sup>	1 <sup>4</sup>	2 <sup>4</sup>
B0a		0 <sub>in</sub> <sup>0</sup>	p1	p2	p3	p4	0 <sub>out</sub> <sup>0</sup>			0 <sub>in</sub> <sup>2</sup>	p1	p2	p3	p4	0 <sub>out</sub> <sup>2</sup>			0 <sub>in</sub> <sup>4</sup>	...
B0b						0 <sub>in</sub> <sup>1</sup>	p1	p2	p3	p4	0 <sub>out</sub> <sup>1</sup>			0 <sub>in</sub> <sup>3</sup>	p1	p2	p3	p4	0 <sub>out</sub> <sup>3</sup>
B1a			1 <sub>in</sub> <sup>0</sup>	p1	p2	p3	p4	1 <sub>out</sub> <sup>0</sup>			1 <sub>in</sub> <sup>2</sup>	p1	p2	p3	p4	1 <sub>out</sub> <sup>2</sup>			1 <sub>in</sub> <sup>4</sup>
B1b							1 <sub>in</sub> <sup>1</sup>	p1	p2	p3	p4	1 <sub>out</sub> <sup>1</sup>			1 <sub>in</sub> <sup>3</sup>	p1	p2	p3	p4
B2a				2 <sub>in</sub> <sup>0</sup>	p1	p2	p3	p4	2 <sub>out</sub> <sup>0</sup>			2 <sub>in</sub> <sup>2</sup>	p1	p2	p3	p4	2 <sub>out</sub> <sup>2</sup>		
B2b								2 <sub>in</sub> <sup>1</sup>	p1	p2	p3	p4	2 <sub>out</sub> <sup>1</sup>			2 <sub>in</sub> <sup>3</sup>	p1	p2	p3
B3a					3 <sub>in</sub> <sup>0</sup>	p1	p2	p3	p4	3 <sub>out</sub> <sup>0</sup>			3 <sub>in</sub> <sup>2</sup>	p1	p2	p3	p4	3 <sub>out</sub> <sup>2</sup>	
B3b									3 <sub>in</sub> <sup>1</sup>	p1	p2	p3	p4	3 <sub>out</sub> <sup>1</sup>			3 <sub>in</sub> <sup>3</sup>	p1	p2

**Figure V.2 Timing Diagram: Optimal Throughput**

Figure V.3 again illustrates a constant stride of one address pattern with an interleaved memory system with a memory ratio of four. In this instance however, there are only three effective banks labeled B0x through B2x where the x is either an a, b, or a c indicating the three cache elements for each bank.

The first 17 cycles are shown in Figure V.3a. The first memory request of each bank is accepted and processed in the same manner as in Figure V.2. However, the second memory request of bank B0 represented by 0<sup>1</sup> is accepted during the third processing cycle (p3) of the previous memory request rather than on the fourth cycle as in Figure V.2. This requires the second memory request to wait one cycle (indicated by the w in cycle five) for the bulk store memory to become available. This scenario is repeated

for each bank (see  $1^1$  and  $2^1$ ). The result is that each bank incurs one wait state. The next set of memory requests labeled  $0^2$ ,  $1^2$ , and  $2^2$  result in a second wait state. In general, an additional wait state is added after each additional memory request is processed until a total of six wait states have been accumulated (see  $0^i$ ,  $1^i$ , and  $2^i$ , for  $i=1\dots6$ ). Memory request  $0^7$  cannot be accepted on cycle 22 since all cache elements are in use. Cache element  $b$  of bank 0 becomes free on cycle 23, freeing cache element  $c$ .

Once the cache elements become saturated as described above, each cache element is associated with a memory request that is either waiting to be processed or is in process. Using  $0^8$  as an example, four processing cycles are used to process  $0^6$ ,  $0^7$ , and  $0^8$ , each requiring four cycles. Therefore, the maximum latency can be seen to be equal to the number of cache elements plus one, times the memory ratio minus the number of effective banks. This relationship is expressed as

$$L_{max} = (NoCE + 1)MR - B_{eff} \quad (V.8)$$

and is applicable to both STM and to standard interleaving.

Following the transient, which ends at cycle five, each set of four cycles (e.g., six through nine) contains three outputs (one from each bank) and one cycle with no output. This yields the anticipated 0.75 throughput as specified in Equation (V.4).

The next section will describe the theoretical memory performance for radix- $r$  butterfly address patterns.

Clk	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
B1a	0 <sup>0</sup>	1 <sup>0</sup>	2 <sup>0</sup>	0 <sup>1</sup>	1 <sup>1</sup>	2 <sup>1</sup>	0 <sup>2</sup>	1 <sup>2</sup>	2 <sup>2</sup>	0 <sup>3</sup>	1 <sup>3</sup>	2 <sup>3</sup>	0 <sup>4</sup>	1 <sup>4</sup>	2 <sup>4</sup>	0 <sup>5</sup>	1 <sup>5</sup>	2 <sup>5</sup>
B0a	0 <sup>0</sup> <sub>in</sub>	0 <sup>0</sup> <sub>in</sub>	p1	p2	p3	p4	0 <sup>0</sup> <sub>out</sub>	0 <sup>2</sup> <sub>in</sub>	w	w	p1	p2	p3	p4	0 <sup>2</sup> <sub>out</sub>		0 <sup>5</sup> <sub>in</sub>	w
B0b					0 <sup>1</sup> <sub>in</sub>	w	p1	p2	p3	p4	0 <sup>1</sup> <sub>out</sub>			0 <sup>4</sup> <sub>in</sub>	w	w	w	w
B0c											0 <sup>3</sup> <sub>in</sub>	w	w	w	p1	p2	p3	p4
B1a			1 <sup>0</sup> <sub>in</sub>	p1	p2	p3	p4	1 <sup>0</sup> <sub>out</sub>	1 <sup>2</sup> <sub>in</sub>	w	w	p1	p2	p3	p4	1 <sup>2</sup> <sub>out</sub>		1 <sup>5</sup> <sub>in</sub>
B1b						1 <sup>1</sup> <sub>in</sub>	w	p1	p2	p3	p4	1 <sup>1</sup> <sub>out</sub>			1 <sup>4</sup> <sub>in</sub>	w	w	w
B1c												1 <sup>3</sup> <sub>in</sub>	w	w	w	p1	p2	p3
B2a				2 <sup>0</sup> <sub>in</sub>	p1	p2	p3	p4	2 <sup>0</sup> <sub>out</sub>	2 <sup>2</sup> <sub>in</sub>	w	w	p1	p2	p3	p4	2 <sup>2</sup> <sub>out</sub>	
B2b							2 <sup>1</sup> <sub>in</sub>	w	p1	p2	p3	p4	2 <sup>1</sup> <sub>out</sub>			2 <sup>4</sup> <sub>in</sub>	w	w
B2c													2 <sup>3</sup> <sub>in</sub>	w	w	w	p1	p2

a)

Figure V.3 Timing Diagram: Non-Optimal Throughput

Clk	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
Bus	1 <sup>5</sup>	2 <sup>5</sup>	0 <sup>6</sup>	1 <sup>6</sup>	2 <sup>6</sup>	0 <sup>7</sup>	0 <sup>7</sup>	1 <sup>7</sup>	2 <sup>7</sup>	0 <sup>8</sup>	0 <sup>8</sup>	1 <sup>8</sup>	2 <sup>8</sup>										
B0a	0 <sup>5</sup> <sub>in</sub>	w	w	w	w	w	p1	p2	p3	p4	0 <sup>5</sup> <sub>out</sub>	0 <sup>8</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	0 <sup>8</sup> <sub>out</sub>
B0b	w	w	p1	p2	p3	p4	0 <sup>4</sup> <sub>out</sub>	0 <sup>7</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	0 <sup>7</sup> <sub>out</sub>				
B0c	p3	p4	0 <sup>3</sup> <sub>out</sub>	0 <sup>6</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	0 <sup>6</sup> <sub>out</sub>								
B1a		1 <sup>5</sup> <sub>in</sub>	w	w	w	w	w	p1	p2	p3	p4		1 <sup>8</sup> <sub>in</sub>	...									
B1b	w	w	w	p1	p2	p3	p4		1 <sup>7</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	1 <sup>7</sup> <sub>out</sub>			
B1c	p2	p3	p4		1 <sup>6</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	1 <sup>6</sup> <sub>out</sub>							
B2a	2 <sup>2</sup> <sub>out</sub>		2 <sup>5</sup> <sub>in</sub>	w	w	w	w	w	p1	p2	p3	p4	2 <sup>5</sup> <sub>out</sub>	2 <sup>8</sup> <sub>in</sub>	...								
B2b	w	w	w	w	p1	p2	p3	p4	2 <sup>4</sup> <sub>out</sub>	2 <sup>7</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	2 <sup>7</sup> <sub>out</sub>		
B2c	p1	p2	p3	p4	2 <sup>3</sup> <sub>out</sub>	2 <sup>6</sup> <sub>in</sub>	w	w	w	w	w	w	p1	p2	p3	p4	2 <sup>6</sup> <sub>out</sub>						

b)

Figure V.3 Timing Diagram: Non-Optimal Throughput (cont.)

## B. RADIX- $R$ BUTTERFLY ADDRESSING

The set of fast Fourier transform algorithms known as the Cooley-Tukey algorithms [Ref 52] provide a technique for computing FFTs for vectors of length  $N$  where  $N$  is defined as,

$$N = r_1^{k_1} \cdot r_2^{k_2} \cdot \dots \cdot r_n^{k_n}, \quad (\text{V.9})$$

where  $N$ ,  $r_i$ , and  $k_i$  are all integers. The set of algorithms used in this architecture are derived using decimation-in frequency [Ref 51]. A derivation of a radix-4 butterfly decimation-in frequency algorithm can be found in Chapter 0.

There are three types of address patterns related to constant geometry fast Fourier transform (FFT). Figure V.4 depicts an eight point decimation-in-frequency FFT using radix-2 butterflies and will be used to illustrate the address patterns related to the computation of FFTs of interest in this dissertation.

To initialize the input data vector, the input data must be placed in sequential order which requires a constant stride pattern of stride one. The analysis for this pass is described in the previous section.

The input address pattern for each intermediate pass is constructed by partitioning the input array into  $r$  parts where  $r$  is the radix of the butterfly. The first element of each partition is accessed to compute the first butterfly. The second element of each partition is then accessed for the second butterfly, etc., until all points of the array have been used. This results in an address pattern of constant stride for each radix- $r$  butterfly. The stride is:

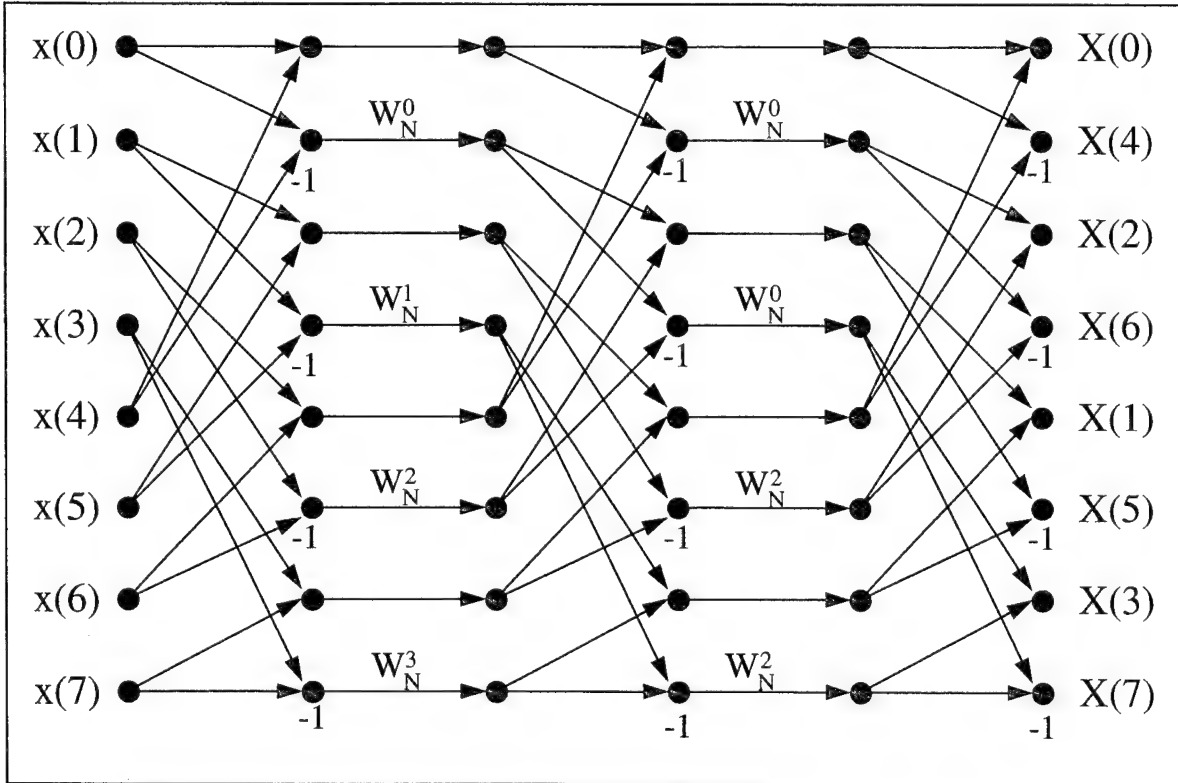
$$S = \frac{N}{r} \quad (\text{V.10})$$

where

$N$  is the length of the input vector (eight in the example), and  
 $r$  is the radix of the butterfly (two in the example).

These relatively short sequences of length  $r$  are concatenated to form the radix- $r$  memory reference stream for a pass. This memory reference pattern is the focus in this section.

The last memory address pattern is digit reversal which is required to access the results of the last pass, as can be seen in Figure V.4. Performance analysis of digit reversal patterns will be addressed in the next section.



**Figure V.4 Radix-2 Constant Geometry Decimation-in-Frequency FFT**

The significance of the above discussion to STMs is that data is selected with a stride  $S$  as defined in Equation (V.10). For a STM with a memory composed of  $B$  banks, up to  $B_{sel}$  banks will be selected where  $B_{sel}$  can be expressed as

$$B_{sel} = \frac{B}{\gcd(S, B)}. \quad (V.11)$$

But, since only the first  $r$  elements of  $B_{sel}$  are taken for the butterfly operation, the actual number of banks selected within a set of numbers for one butterfly can never be greater than  $r$ .

On the other hand, since the address used in each partition increases by one for each set of numbers used in a butterfly operation, all banks will be used. In the worst case,  $B$  is a factor of  $S$ , and  $B_{sel} = 1$ . In this case, a single bank will receive all  $r$  of the memory requests for a given butterfly. However, the next set of  $r$  numbers taken for the butterfly, will be sent to another bank. All banks will be given a task within  $B$  sets of butterflies or within  $r \cdot B$  samples. If the number of banks and the radix are both powers of two, this worst case scenario is the case.

The steady-state throughput and maximum latency can be visualized for standard interleaving with the aid of Figure V.5 which contains a segment of a timing diagram for a radix-4 butterfly address pattern for a standard interleaved memory with four banks. Note only three banks are shown.

Clk	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Bus	0 <sup>2</sup>	0 <sup>3</sup>	0 <sup>3</sup>	0 <sup>3</sup>	0 <sup>3</sup>	1 <sup>0</sup>	1 <sup>1</sup>	1 <sup>1</sup>	1 <sup>1</sup>	1 <sup>1</sup>	1 <sup>2</sup>	1 <sup>2</sup>	1 <sup>2</sup>	1 <sup>2</sup>	1 <sup>3</sup>	1 <sup>3</sup>	1 <sup>3</sup>	1 <sup>3</sup>	2 <sup>0</sup>	
B0		0 <sup>2</sup> <sub>in</sub>	p1	p2	p3	p4	0 <sup>2</sup> <sub>out</sub>													
B1			p4	0 <sup>1</sup> <sub>out</sub>		0 <sup>3</sup> <sub>in</sub>	p1	p2	p3	p4	0 <sup>3</sup> <sub>out</sub>									
B2						1 <sup>0</sup> <sub>in</sub>	p1	p2	p3	p4	1 <sup>0</sup> <sub>out</sub>				1 <sup>2</sup> <sub>in</sub>	p1	p2	p3	p4	1 <sup>2</sup> <sub>out</sub>
										1 <sup>1</sup> <sub>in</sub>	p1	p2	p3	p4	1 <sup>1</sup> <sub>out</sub>				1 <sup>3</sup> <sub>in</sub>	p1
																				2 <sup>0</sup> <sub>in</sub>

**Figure V.5 Timing Diagram for Radix-4 Butterfly Pattern (Standard Interleaving)**

First consider the steady-state throughput. It is assumed in this diagram that only one bank is selected for each butterfly as described above. In this case, a bank  $B_i$  will receive  $r$  consecutive memory requests followed by  $r$  memory requests to bank  $B_{i+1}$ . When the last bank receives its memory requests, the process repeats with the first bank. From the figure, it can be seen that the pattern is cyclic and contains two regions of activity as it relates to throughput. For a given bank, the first and last memory references are processed in parallel with the previous and next banks, respectively. This is the first type of region referred to above. One such region is located between cycles six through ten, and the next between cycles 19 through 23 (Only cycles 18 and 19 are shown in the figure). The other type of region is found between instances of the first type of region. In this second type of region, one bank is processing memory requests alone (i.e., no other

banks have requests to process). The representative region shown in the figure is between cycles 11 through 18. Therefore, one representative period of this cyclic pattern begins with cycle six and ends with cycle 18. Since the pattern is cyclic, the throughput represented by this period is the steady-state throughput.

The number of cycles represented by the two regions is  $MR+1$  and  $(r-2)MR$ , respectively, for a total of  $MR+1+(r-2)MR$  cycles. During this period of time, two outputs occur in the first region and  $r-2$  outputs occur in the second region for a total of  $r+2$  outputs. The steady-state throughput is the ratio of the number of outputs to the total number of cycles during the period and is expressed as

$$\begin{aligned} TP_{ss} &= \frac{r}{MR+1+(r-2)MR} \quad \text{when } B \geq MR \\ &= \frac{r}{(r-1)MR+1}. \end{aligned} \tag{V.12}$$

This analysis assumes that the number of banks is matched to the memory ratio.

The maximum latency can be determined by inspection of Figure V.5. Consider one memory request such as  $1^1$ . It is available initially on the bus at cycle six and must first wait for  $1^0$  to finish processing. This results in a delay of  $MR$  cycles. Processing of  $1^1$  requires  $MR$  more cycles. One additional cycle is needed to transfer the result back to the processor for a total maximum latency

$$L_{max} = 2MR + 1 \tag{V.13}$$

when  $B \geq MR$ .

Now consider STM memory architectures. As indicated above, all banks are utilized in radix- $r$  address patterns. Further, a maximum of  $r$  consecutive memory requests can be made to a bank. Therefore, the banks will be utilized efficiently if

$$NoCE \geq r+1 \tag{V.14}$$

because this ensures that a bank will not stall when presented with  $r$  consecutive memory requests. The steady-state throughput is then



$$TP_{ss} = \frac{B}{MR} \text{ when } NoCE \geq r+1 \quad (V.15)$$

for all STM cases since all banks will be effectively engaged for a radix- $r$  pattern.

The latency for a STM memory will be described with the aid of Figure V.6. A total of  $r \cdot MR$  cycles are required to process all of the  $r$  memory requests sent to a bank. The last memory request must wait for the others to finish. This analysis assumes that the number of banks is sufficient to obtain optimum throughput and represents the maximum latency for the address pattern. The last memory request is not available to the memory system until  $r-2$  cycles, with respect to the first memory request. Further, an additional cycle is needed to return the memory response to the processor. The maximum latency is therefore

$$\begin{aligned} L_{max} &= r \cdot MR - (r-2) + 1 \quad \text{when } B \geq MR, NoCE \geq r+1 \\ &= r(MR-1) + 3 \end{aligned} \quad (V.16)$$

for STM memories.

Clk	0	1	2	3	4	5	6	7	8	9	10	11	12
Bus	0 <sup>0</sup>	0 <sup>1</sup>	0 <sup>2</sup>	0 <sup>3</sup>	...								
B0a		0 <sup>0</sup> <sub>in</sub>	p1	p2	p3	p4	0 <sup>0</sup> <sub>out</sub>						
B0b			0 <sup>1</sup> <sub>in</sub>				p1	p2	p3	p4	0 <sup>1</sup> <sub>out</sub>		
B0c				0 <sup>2</sup> <sub>in</sub>							...		
B0d					0 <sup>3</sup> <sub>in</sub>							...	0 <sup>3</sup> <sub>out</sub>

**Figure V.6 Timing Diagram for Radix-4 Butterfly Pattern STM(4,5,4)**

### C. DIGIT REVERSAL

An address can be expressed as

$$\text{index} = a_n r^n + a_{n-1} r^{n-1} + \dots + a_2 r^2 + a_1 r + a_0 \quad (V.17)$$

where

$r^i$  is the radix of the butterfly operation raised to the  $i$ th power,

$a_i$  is a digit of the base  $r$  number system, and

index is the index into the data array. It will be assumed that the array begins at address 0, making the index equivalent to the address. This is a valid assumption since shifting the array in the address space does not effect the analysis.

The equivalent digit-reversed number representation is then

$$\text{index}_{\text{dr}} = a_0 r^n + a_1 r^{n-1} + a_2 r^{n-2} + \cdots + a_{n-1} r + a_n \quad (\text{V.18})$$

where  $\text{index}_{\text{dr}}$  is the digit-reversed index.

The digit-reversed address pattern is a constant stride pattern with a stride of one that is digit reversed. The resulting sequence is one that increments by  $r^n$  as  $a_0$  cycles from 1 to  $r-1$ . When  $a_0 = 0$ ,  $a_1$  increments. The relationship holds for  $a_i$  and  $a_{i-1}$  for each  $i$ .

Therefore, it can be seen that the digit reversal address pattern is composed of a set of constant stride sequences of length  $r$  that are concatenated together. Equation (V.1) provides insight into the effectiveness of an interleaved memory system with conventional decoding. Within a sequence, the effective number of banks is

$$B_{\text{eff}} = \frac{B}{\text{gcd}(r^k, B)}. \quad (\text{V.19})$$

If  $r$  and  $B$  are relatively prime, then  $B_{\text{eff}} = B$  for each sequence and for the address pattern at large. If, however, both the number of banks and the radix is a power of two, then the effective number of banks is one for all practical situations. Therefore, when the number of banks and the radix are a power of two, the throughput approaches  $1/B$ .

This result is based on the assumption that the number of cache elements is relatively small with respect to the length of the input vector. Suppose that the number of cache elements is sufficiently large to accept all memory requests without a stall. The digit-reversed address pattern has the property that each bank receives  $N/B$  consecutive memory requests, where  $N$  is the length of the input vector and  $B$  is the number of banks. Because each bank has a sufficient number of cache elements ( $N/B$ ) to accept all of the

memory requests without a stall, all memory requests are delivered in  $N$  cycles. The last bank receives its first memory request at cycle

$$N - \frac{N}{B} = \frac{NB - N}{B}. \quad (\text{V.20})$$

Assuming that the number of banks is matched to the memory ratio, the number of cycles required for the last bank to process its memory requests is

$$\frac{N}{B} MR = N. \quad (\text{V.21})$$

Therefore, the number of cycles required to process all of the memory requests is

$$\frac{NB - N}{B} + N = \frac{NB - N + NB}{B} = \frac{2NB - N}{B}. \quad (\text{V.22})$$

The average throughput is defined as the ratio of the number of cycles needed with an ideal memory device to the actual number of cycles required (see Equation II.4) therefore

$$TP = \frac{N}{\frac{2NB - N}{B}} = \frac{B}{2B - 1} \quad (\text{V.23})$$

when  $NoCE \geq N/B$ . Although this represents a substantial improvement from the previous result, it is achieved at a substantial cost in hardware. In any case, it provides a throughput of approximately 0.5 for even a modest number of banks.

The poor performance of an interleaved memory system using conventional decoding for the digit-reversed case, strongly suggests that a modification is required in order to obtain satisfactory throughput for the digit reversal pattern when that base of the digit is a power of two. The modification selected is permutation-based memory decoding as described in Section E of Chapter 0. The following discussion describes the anticipated performance for constant stride, radix-2, and digit-reversed address patterns when permutation-based decoding is used.

#### D. PERMUTATION-BASED DECODING PERFORMANCE

In this section, the performance of the three memory address patterns described above will be analyzed based on a bank decoding scheme using a permutation matrix as described in Section E of Chapter 0. Following the approach above for conventional decoding, the simplest addressing pattern, addresses with constant stride will be analyzed first. Results from this analysis will then be applied to the radix- $r$  butterfly and digit-reversed addressing patterns.

Permutation based decoding was pursued due to the poor performance encountered when the number of banks in the memory system and the characteristic of the addressing pattern (e.g., the stride in constant stride addressing patterns) were not relatively prime. The problem is most severe for digit-reversed patterns that are characterized with sequences with large constant strides.

As shown in Chapter 0, Section E, each bank is selected once and only once within a base sequence when a non singular permutation matrix with dimension  $n$  by  $n$  is used to decode the bank number. An expanded permutation matrix that uses more address bits for bank selection results in the base sequence of bank numbers to be permuted as illustrated in Figure III.12. All of the bank numbers are represented in each block although the order will usually vary.

Therefore, the worst case scenario is that the last bank number of one block will be followed by the same bank number of another block. For example, for a four-bank memory, the first and second blocks could be  $\{0\ 1\ 2\ 3\}$  and  $\{3\ 2\ 1\ 0\}$  respectively. If these were the only permutations of the bank number pattern, the banks 3 and 1 would always be given two consecutive memory references.

The following describes the steady-state throughput and maximum latency when permutation-based bank decoding is in use. In a standard interleaved memory, the lower bound of the steady-state throughput can be derived by observing a cyclic pattern of the output. Within a set of bank numbers, each bank receives a request, processes the request, and then places its response on the bus at the appropriate time. The memory system responds with a total of  $MR$  outputs. Since the first processing cycle for the

second request of the last bank occurs as the last bank sends the first output back to the processor, there will be  $MR-1$  cycles with no output. Therefore, there are  $MR$  cycles with output followed by  $MR-1$  cycles with no outputs. Since the banks accept and process the requests of the second set with no further delay (after the second memory request of the last bank is accepted) then  $MR$  outputs occur following the  $MR-1$  period of no outputs. At this point, two consecutive memory requests are encountered by the first bank and the pattern repeats. Therefore, for a standard interleaved memory system, the worst case steady-state throughput is

$$TP_{ss}^{lb} \geq \frac{MR}{MR + (MR - 1)} = \frac{MR}{2MR - 1} \text{ for } B \geq MR. \quad (V.24)$$

Under these circumstances, the upper bound of the maximum latency is incurred by the second consecutive memory request to a bank. This memory request must first wait for the preceding memory request to be processed ( $MR$  cycles), followed by  $MR$  cycles to process this memory request, and finally a cycle to return the memory response. Therefore, the upper bound for the maximum latency for constant-stride address patterns for standard interleaving is

$$L_{max}^{ub} \leq 2MR + 1 \text{ for } B \geq MR. \quad (V.25)$$

Since all of the banks are utilized, a STM with three or more cache elements will provide sufficient buffering to ensure full utilization of all of the banks. Therefore, the steady-state throughput for a STM memory is

$$TP_{ss} = \frac{B}{MR} \text{ for } B \leq MR \text{ and } NoCE \geq 3. \quad (V.26)$$

The latency for STM memories is the same as for standard interleaving for constant-stride address patterns, given that the number of banks is matched to the memory ratio and the number of cache elements is three or more. The only difference between standard interleaving and a STM memory is that the second memory request is not accepted by the memory in the standard interleaving case until the last processing cycle, whereas the STM memory will accept it when the request first appears on the bus (i.e., the memory request is not accepted for first  $MR$  cycles in standard interleaving but is

accepted by STM) In either case, there are  $MR$  cycles required to process the first request followed by  $MR$  cycles to process the second request and a cycle to return the processed memory response. Therefore, the upper bound of the maximum latency for constant stride patterns for both the standard interleaving as well as for STM memories is

$$L_{max}^{ub} \leq 2MR + 1 \text{ for } B \geq MR, \text{ and } NoCE \geq 3. \quad (V.27)$$

Now consider the use of permutation-based decoding for a radix- $r$  butterfly address stream. The radix- $r$  butterfly addressing pattern provides a unique bank for each of the inputs to a single radix- $r$  butterfly calculation because this is a sequence of constant stride ( $s = N/r$ ) as long as the number of banks is greater than the radix- $r$ . If the number of banks is less than  $r$ , the bank numbers will repeat and there exists the possibility that two consecutive bank numbers can occur when crossing over a block boundary. This situation is similar to the constant stride case where the last bank of one base set can be the first bank in the next set. Clearly if the radix is smaller than the number of banks, then only a subset of the banks will be selected.

The major concern for radix- $r$  butterfly address patterns when using permutation-based bank decoding is the relationship between the sets of banks selected for the butterfly operations. This address pattern can be viewed as an interleaving of  $r$  streams of constant stride of one address pattern. The effect of this  $r$ -way interleaving is not clear, given the current set of constraints on the address stream, namely that it consists of a sequence of blocks where each block contains a permutation of the bank numbers. The larger the value of  $r$ , the greater the potential impact to the desired properties of the address stream.

To clarify this last point, note that for a radix-2 butterfly, two constant-stride address patterns with a stride of one are interleaved. Suppose the number of banks is eight. The following is an example of the problems possible with radix-2 addressing:

Sequence #1: {1, 2, 3, 4 ...}

Sequence #2 {2, 3, 4, 5 ...}

with a resulting sequence of  $\{1, 2, 2, 3, 3, 4, 4, 5 \dots\}$ . A worse case scenario occurs if both sequences are on a boundary with a repeating bank number resulting in one bank number four consecutive times. As the radix increases, the potential for disrupting the desired pattern increases.

An alternative to this dilemma is to construct a permutation matrix that has properties favorable for radix- $r$  addressing patterns. The following discussion will describe one technique for constructing such a matrix.

The matrices designed with the technique described below are tailored both to the number of banks as well as to the stride  $s$ . The use of tailored matrices requires that the permutation matrix be loaded prior to using the memory. The permutation matrix cannot be changed until the data inside the memory is not required anymore. A review of Figure III.9 indicates that a memory engaged in the radix- $r$  pattern will also be required to accept a constant stride pattern with a stride of one. Therefore, the constraints necessary to ensure good performance for constant-stride address patterns will also be applied to the matrices designed for radix- $r$  patterns.

The following description for constructing permutation-based matrices for radix- $r$  address patterns will use a STM that has 16 banks to illustrate the construction process. Other STM configurations can readily be constructed by applying the principles described below.

Figure V.7 illustrates the desired mapping to the address space when the stride of the radix- $r$  butterfly is 16 by a permutation matrix to be described below. The address space is represented by the matrix with column order (i.e., the first 16 elements of the address space are represented by the first column. The contents of each element is its bank number. For simplicity, the first 16 elements are mapped with the identity matrix as indicated in the figure and the permutation matrix shown in the following equation

$$\begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{a}. \quad (\text{V.28})$$

Base	M1 p <sub>0,5</sub>	M2 p <sub>1,4</sub>		M3 p <sub>2,3</sub>				M4 p <sub>3,2</sub>							
0000	1000	0100	1100	0010	1010	0110	1110	0001	1001	0101	1101	0011	1011	0111	1111
0001	1001	0101	1101	0011	1011	0111	1111	0000	1000	0100	1100	0010	1010	0110	1110
0010	1010	0110	1110	0000	1000	0100	1100	0011	1011	0111	1111	0001	1001	0101	1101
0011	1011	0111	1111	0001	1001	0101	1101	0010	1010	0110	1110	0000	1000	0100	1100
0100	1100	0000	1000	0110	1110	0010	1010	0101	1101	0001	1001	0111	1111	0011	1011
0101	1101	0001	1001	0111	1111	0011	1011	0100	1100	0000	1000	0110	1110	0010	1010
0110	1110	0010	1010	0100	1100	0000	1000	0111	1111	0011	1011	0101	1101	0001	1001
0111	1111	0011	1011	0101	1101	0001	1001	0110	1110	0010	1010	0100	1100	0000	1000
1000															
1001															
1010															
1011															
1100															
1101															
1110															
1111															

.

.

.

Figure V.7 Required Mappings When the Stride Equals the Number of Banks



Note that only the first eight rows of the matrix are filled in addition to the first column, and that the bank numbers are in binary. By inspection of Equation (V.28), the permutation matrix consist of the identity matrix (four rightmost columns) preceded by four columns that will now be discussed. The first mapping, labeled M1 in Figure V.7, is a result of  $p_{0,5} = 1$  in Equation (V.28) followed (in the order in which they are applied to the sequence) by  $p_{1,4} = 1$ ,  $p_{2,3} = 1$ , and  $p_{3,2} = 1$ , labeled M2 through M4 respectively. The matrix is zero indexed with the origin in the upper left-hand corner.

As indicated before, this matrix is designed for a stride of 16. Assuming that the first element of the sequence is at the origin, the sequence making up this stride of 16 consists of a row-wise ordering of the matrix. Thus far, the radix of the butterfly has not been specified. Suppose first that  $r=16$ . In this case, the first butterfly operation will receive the first row of the matrix in Figure V.7; the second butterfly operation will receive the second row, etc. In this instance, it can be seen that the effect of mappings M1 through M4 is to permute the first element in a row to all possible bank numbers. Therefore, since the matrix will be accessed in row order, an address stream is generated that has the same properties as that of an address stream with a constant stride of 16. The resulting performance of this radix- $r$  address stream should be consistent with a constant stride addressing stream described above.

Suppose now that the radix is not 16, but rather two, four, or eight (these are the radices of interest in this effort). For any other radix, the addressing pattern remains row wise. However, only the first  $r$  elements of the matrix are taken for each butterfly operation. Assuming that the first reference is at the upper left-hand corner of the matrix, the addresses for the first butterfly operation is the first  $r$  elements of the first row. The next butterfly operation uses the first  $r$  elements in the second row, etc.

Consider first a radix-2 butterfly address stream. The M1 mapping results in the selection of each bank after eight butterfly operations or 16 memory references. This is accomplished by the M1 mapping by toggling the  $b_3$  bank bit for the base sequence. This maps elements 0000 through 0111 to the second half of the 0000 through 1111 sequence, thereby ensuring all sixteen banks are accessed with the radix-2 pattern. An inspection of

Figure V.7 will verify that similar statements hold for radix-4 and 8 sequences which also require the M2 and M3 mappings respectively, to get the desired result.

Radix- $r$  sequences with strides longer than sixteen will take advantage of the permutation matrix elements to the left of those used to implement mappings M1 through M4 ( $p_{0,2}$  and  $p_{1,3}$  are shown in Equation (V.28)). The effect of these mappings is to map the results of M1 through M4 to other permutations. However, the relationship between the banks is preserved through these mappings, and therefore the desired properties are preserved.

Recall that one of the requirements for these matrices is that they meet the conditions required for constant stride matrices. In particular, all sub-matrices of the permutation matrix of dimension  $n$  by  $n$ , where  $n$  is the number of bits required to represent the bank number, must be nonsingular. In Equation (V.28), this is clearly not the case because each row contains a string of four or more zeros. This can be easily fixed however by inserting zeros at positions  $p_{0,1}$ ,  $p_{1,1}$ ,  $p_{6,2}$ , and at  $p_{7,2}$ , which satisfies the requirements for constant stride matrices while maintaining the requirements for radix- $r$  matrices.

Two additional situations must be addressed: when the stride is less than the number of banks and when the stride is greater than the number of banks. First, suppose that the stride is less than the number of banks. For example, if the stride were a half of the number of banks, the first and the eight elements of the base sequence address would be accessed. In this situation, the permutation matrix needs to transform these two elements to the remaining elements. Using a similar strategy as that above, three mappings, M1, M2, and M3 map the first half of the base sequence into the first row and the second half of the base sequence into row eight, as shown in Figure V.8 using the permutation matrix of Equation (V.29).

$$\begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{a}. \quad (\text{V.29})$$

A radix-16 sequence of banks is the interleaving of the first and eighth rows {0000, 1000, 0100, 1100, 0010, 1010, ... 0111, 1111} for the first radix. The second radix operation has a similar pattern for the second and ninth rows.

Observe that a radix-2 pattern will produce the sequence of bank numbers {0000, 1000, 0001, 1001, 0010 ...} and that the radix-4 sequence results in {0000, 0100, 1000, 1100, 0001, 0101, 1001, 1101, 0010 ...}. Observe that in each case, each bank number is encountered once every 16 memory references.

Base	M1 $p_{1,5}$	M2 $p_{2,4}$		M3 $p_{3,3}$			
0000	0100	0010	0110	0001	0101	0011	0111
0001	0101	0011	0111	0000	0100	0100	0110
0010							
0011							
0100							
0101							
0110							
0111							
1000	1100	1010	1110	1001	1101	1011	1111
1001	1101	1011	1100	1000	1100	1010	1101
1010							
1011							
1100							
1101							
1110							
1111							

**Figure V.8 Mapping Required When Stride is One Half the Number of Banks**

The last situation is when the stride is greater than the number of banks. Only the first element of the base sequence is referenced (as in the case when the stride was equal to the number of banks). The required sequence of mappings M1 through M4 is shown in Equation (V.28). However, this mapping must be shifted to the left within the matrix. For example, if the stride is two times the number of banks, the mappings M1 through M4 must be shifted one position to the left (four times two positions, etc.). Such a matrix

is shown in Equation (V.30) for a stride of four times the number of banks when the number of banks is 16. The two columns between the identity matrix and the mappings provide the necessary shifting of the mapping matrix.

$$\begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \mathbf{a}. \quad (\text{V.30})$$

In summary, a radix- $r$  addressing pattern requires tailored matrices to yield satisfactory performance for radix values greater than two. The matrices must be tailored to the stride of the radix- $r$  address pattern. In general, the three cases that must be taken into account are when the stride is less than, equal to, or greater than the number of banks. When these matrices are used, the performance of the radix- $r$  address patterns are equivalent to those of constant stride with respect to steady-state throughput and maximum latency. The next section will address digit-reversed address patterns when permutation-based bank decoding is used.

The steady-state throughput for a digit-reversed pattern is primarily governed by maximum stride which is equal to the place value of the most significant digit of the address of the input vector. This stride is repeated  $r$  times where  $r$  is the radix of the butterfly used to compute the fast Fourier transform. These constant stride sequences of length  $r$  and stride  $r^{k-1}$ , where  $k$  is the number of digits in the address, are concatenated together to form the digit-reversed pattern.

Permutation based decoding will ensure that the banks selected within a constant stride sequence will be unique up to the number of banks. If the radix is equal to or greater than the number of banks, then each set of constant stride sequences will contain an equal number of references to each bank. This will yield a steady-state throughput and maximum latency consistent with constant stride addresses.

If the radix is smaller than the number of banks, then unique banks will be selected within each  $r$  length sequence. However the relationship between the bank

numbers between sequences is not known. Therefore, the same banks could be selected again.

The large number of permuted patterns suggested by Figure III.12 and the permutation matrices in Figure III.13 through Figure III.15 suggest that a variety of banks can be selected under these circumstances.

The lower bound for the steady-state throughput in this situation is

$$TP_{ss,lb} \geq \frac{r}{MR} \text{ for } r \leq B \text{ and } NoCE \geq 3. \quad (V.31)$$

The maximum latency is governed by the number of cache elements and the memory ratio under these circumstances. The maximum latency is

$$L_{max}^{ub} \leq (NoCE + 1)MR - 1. \quad (V.32)$$

In this chapter, performance of both standard interleaving memories as well as STM memories were analyzed first for conventional memory decoding and then for permutation-based memory decoding. Addressing patterns analyzed include constant stride, radix- $r$  butterfly, and digit-reversed addressing patterns.

Constant-stride address patterns provide optimum performance under conventional decoding when the stride and number of banks is relatively prime. The steady-state throughput is 1.0 and the maximum latency is  $MR + 2$ . However, the architecture in Chapter 0 requires strides that are powers of two. Address streams with these strides perform poorly using conventional decoding as described in Equation (V.4). Performance for constant stride patterns that are not powers of two is not specified based on the theory of the permutation matrices developed.

When constant-stride address patterns with strides of powers of two are applied to a STM memory with permutation-based decoding, the steady-state throughput is optimal and the latency increases to an upper bound of  $2MR + 1$ . This is slightly less than double that incurred with conventional decoding.

Radix- $r$  address patterns yield an optimal steady-state throughput for all radix values ( $r = 2, 4, 8$ , and  $16$ ) but with latencies proportional to the product of the radix and

the memory ratio. Standard interleaving performed very poorly for this case because this pattern for the cases of interest result in  $r$  consecutive hits to the same bank.

Little can be said regarding performance when using the permutation-based matrices for radix- $r$  butterfly patterns. However, when tailored permutation matrices are used for radix- $r$  butterfly patterns, optimal throughput with an upper-bound latency consistent with constant stride patterns (i.e.,  $2MR + 1$ ) are predicted.

Conventional decoding performs poorly for digit reverse address patterns because the digit-reversed patterns of interest are characterized by sequences of length  $r$  constant stride with the stride a power of two. The steady-state throughput is expected to be inversely proportional to the number of banks when the number of cache elements is small. If the number of cache elements is large (i.e.,  $\approx N/B$ ) then the average throughput is

$$\frac{B}{2B-1}$$

where  $B$  is the number of banks. The gain in throughput is obtained by a substantial investment of hardware. Standard interleaving is also expected to perform poorly with a steady-state throughput inversely proportional to the number of banks because this pattern is characterized by long sequences to a single bank.

The permutation-based theoretical results are mixed when applied to digit-reversed address patterns. When the radix is equal to or greater than the number of banks, then the projected performance is consistent with constant-stride address patterns using permutation-based techniques. When the radix is less than the number of banks, a loose lower bound expression for the steady-state throughput is  $r/MR$  and the latency is an upper-bound expression that is proportional to the product of the number of cache elements and the memory ratio.

## E. RANDOM ADDRESSING

As indicated in Equation (II.10), the speedup of a standard interleaved memory system (i.e., one without any buffering) yields a speedup that is approximately the square

root of the number of banks. It is desirable to know the impact of buffering on the performance of the interleaved memory system, given that the input address stream is random.

Queuing theory provides a framework for analyzing this problem. For background information on this topic see Trivedi [Ref 53] and Allen [Ref 54]. The following discussion is based on an address stream consistent with Equation (II.10), developed by Hellerman [Ref 25], that assumes each bank is equally likely to be selected for each memory address issued to a memory system that contains  $MR$  banks. If the problem is modeled in a queuing theory context, each of the banks can be modeled as a queue with a single server (i.e., the bulk storage unit). This server has a service distribution that is deterministic with a constant service cycle time of  $MR$ .

The input rate to the memory system (i.e., all of the banks) is one request per cycle. The equal probability assumption on bank selection results in a geometric distribution for the interarrival time with a mean arrival time of  $1/MR$ , or equivalently  $1/B$  where  $B$  is the number of banks. Given a queue length of  $k$ , a single bank can be described using queuing theory notation as a  $M/D/1/k$  queuing problem where  $M^1$  represents the distribution of the interarrival time,  $D$  is the distribution of the server time, the 1 indicates a single server, and as indicated above, the  $k$  is the queue length of the input queue to the server. At times it may be useful to assume that  $k = \infty$ .

There are several features of this problem that distinguish it from traditional queuing problems. The queue length is finite and there is no *balking* (i.e., a customer does not leave a line no matter how long the customer must wait). Further, since read cycles are assumed, the order in which the requests are made must be preserved across all of the banks. The implication is that a given customer cannot leave the queue until all of the customers that preceded it leave their respective queues. This can lead to nonsensical situations when interpreted as a typical service line for humans. For example, it is possible for a bank to have a full queue of processed customers that are waiting for the

---

<sup>1</sup>  $M$  is generally reserved to represent the exponential distribution, which is the continuous counterpart to the discrete-time geometric distribution.

proper turn to be sent back to the processor. Therefore, even though the queue is full, the server has nothing to process and cannot accept new requests until a serviced customer exits the queue.

A closed-form solution was not obtained for this queuing problem. However, the following observations are made concerning this process. Because the number of banks is matched to the memory ratio, the banks must be fully utilized in order for the service rate to be equal to the input rate. This is possible only if the inputs are assigned in a round robin fashion as described earlier in this section. The random nature of the input stream will certainly not provide this type of assignment and therefore the service rate will be less than the input rate. So long as the input rate exceeds the service rate, the queue length will grow and if the queue length is modeled as infinite, there is no steady-state solution. If, however, finite queue lengths are assumed, then stalls that occur when queues fill up serve to regulate the input and a steady-state condition is obtained.

One experiment will be generated to analyze this problem.



## VI. SIMULATION STUDIES

### A. OVERVIEW

The following section is a description of the split transaction memory (STM) simulations executed for the purposes of this dissertation. The major emphasis of these simulation runs is to verify the analytic results obtained in Chapter V and to provide data for making architectural choices for the vector processor architecture. A secondary goal is to explore the use of STM for general-purpose computing.

The simulation studies are organized into two major groups. The first group is concerned with vector processing. The second group consists of a single experiment focused on general-purpose computing. Input variables pertinent to both groups include the architectural parameters number of bank (**NoBanks**), number of cache elements (**NoCE**), and memory ratio (**MemRatio**). The memory decoding scheme (**MemDecode**) is an important parameter for the vector processor simulations but not those concerning general-purpose computing. The type of address pattern is the another key input to a simulation run. The vector processor simulations are organized by the three address patterns discussed in Chapter V: constant stride, radix- $r$  butterfly, and digit-reversed address patterns. The random address pattern is analyzed for the general-purpose case.

The primary measurements of performance that are analyzed for both simulation groups are the steady-state throughput (**SSTP**) and the maximum latency (**ML**). Note that all of the performance parameters described in Section D of Chapter II are measured during each simulation and are included in the discussion below when appropriate. Speedup is also addressed in the general-purpose computing simulation.

The vector processing experiments are summarized in Table VI.1. They are organized into three pairs, each corresponding to an address pattern. Each pair first addresses conventional memory decoding followed by permutation-based memory decoding.

The first set of experiments deals with constant-stride address patterns. The first experiment is designed to verify the problems associated with using conventional

decoding when the stride and the number of banks are not relatively prime. This experiment also demonstrates optimal STM performance when the stride and the number of banks are relatively prime and thereby places an upper bound on the goodness of STM performance for the remaining experiments.

Name	Purpose	Scope
Constant Stride (conventional decoding)	Verify constant stride analysis for conventional decoding.	Stride = 1, 2, 3, 4, 5, 6, 7, 8, and 9
Constant Stride (PB decoding)	Evaluate the performance of STM using PB for constant-stride address patterns where $s=2^k$ , $k=1,2,..$	Stride = 1, 2, 3, 4, 5, 8, 16, 32, 64, and 128.
Radix- $r$ Butterfly (conventional decoding)	Verify radix- $r$ analysis for conventional decoding.	$r = 2, 4, 8$ , and 16.
Radix- $r$ Butterfly (PB decoding)	Evaluate the performance of STM using PB for radix- $r$ butterfly address patterns.	$r = 2, 4, 8$ , and 16.
Digit Reversal (conventional decoding)	Verify digit-reversed analysis for power of two base number systems using conventional decoding.	$base / NoDigits = 2/10$ , $4/5$ , $8/4$ , and $16/4$ .
Digit Reversal (PB decoding)	Evaluate the performance of STM using PB for digit-reversed address patterns for power of two base number systems.	$base / NoDigits = 2/10$ , $4/5$ , $8/4$ , and $16/4$ .

**Table VI.1 Vector Processor Experiments**

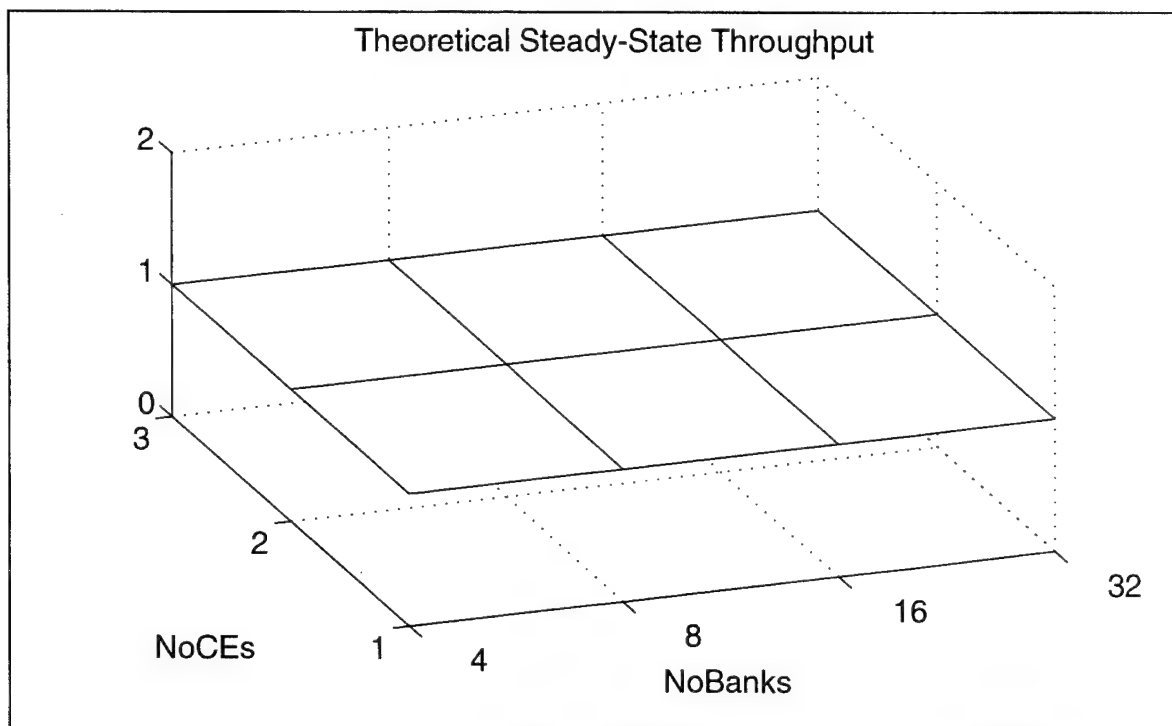
The set of parameters used in the first experiment is:

$$\begin{aligned}
 \text{NoBanks} &= 4, 8, 16, 32 \\
 \text{MemRatio} &= \text{NoBanks} \\
 \text{NoCE} &= 1, 2, 3,
 \end{aligned}
 \tag{VI.1}$$

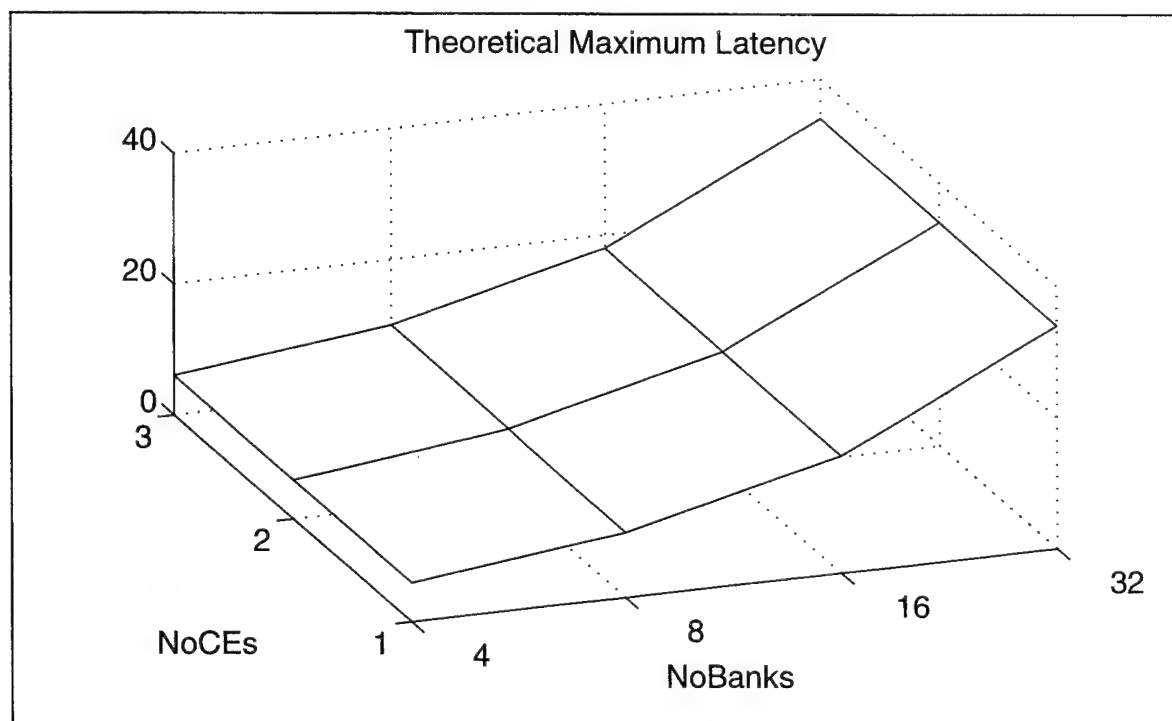
where  $\text{NoCE}=1$  is to be understood as a single buffer in standard interleaving rather than a STM with one cache element. This convention will be assumed hereafter for the following experiments. The values for the number of banks and the corresponding values for the memory ratio are also used in all of the other vector processor experiments. The number of banks is matched to the memory ratio based on the premise that an optimal

throughput is obtainable without increasing the number of banks to obtain throughput. The range of values for the number of cache elements is tailored for each experiment. The standard interleaving case is always provided for comparison ( $\text{NoCE} = 1$ ) to the STM cases. The number of cache elements that is expected to provide an optimum steady-state throughput based on the analysis in Chapter V ( $\text{NoCE} = 2$  for the experiment above) is also included. Additional values for the number of cache elements may be provided to explore the sensitivity of the performance values to the number of cache elements ( $\text{NoCE} = 3$  above).

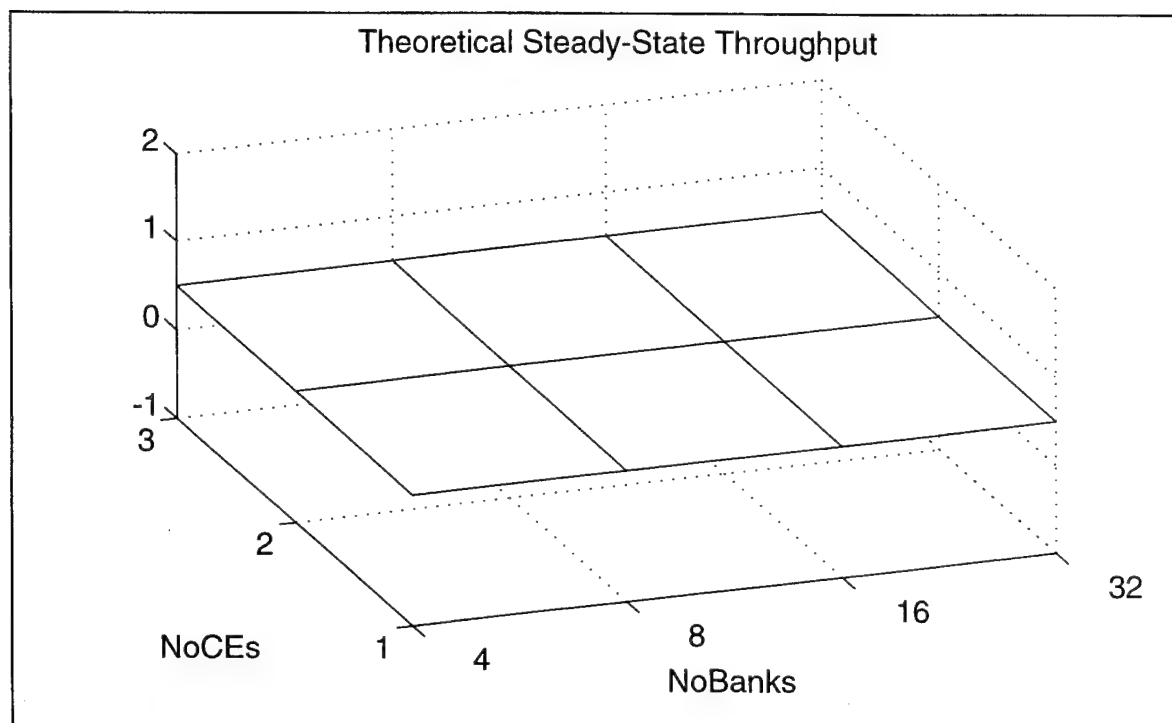
This experiment is designed to validate expressions for the steady-state throughput and latency as expressed in Equations (V.4) through (VI.8). Plots generated based on these equations are shown in Figure VI.1 through Figure VI.8. Figure VI.1 and Figure VI.2 illustrate the steady-state throughput and maximum latency, respectively, for those strides that are relatively prime to the number of banks (i.e., strides 1, 3, 5, 7, and 9). These figures show the best performance possible for an interleaved memory system. Figure VI.3 and Figure VI.4 reflex the throughput and latency for a stride of two. The steady-state throughput is 0.5 for all values because the number of effective banks is half of the total number of banks, which is in turn equal to the memory ratio. A similar relationship holds for a stride of four except the effective number of banks is one fourth of the total number of banks as shown in Figure VI.5. The corresponding maximum latencies for a stride of four are reflected in Figure VI.6. The steady-state throughput is slightly more complicated for a stride of eight because the effective number of banks is a fourth of the total number of banks when the number of banks is four. For the cases where the number of banks is eight and sixteen, the effective number of banks drops to one eighth of the total number as shown in Figure VI.7. This is due to the greatest common denominator operation in Equation (V.1). The corresponding maximum latencies are shown in Figure VI.8.



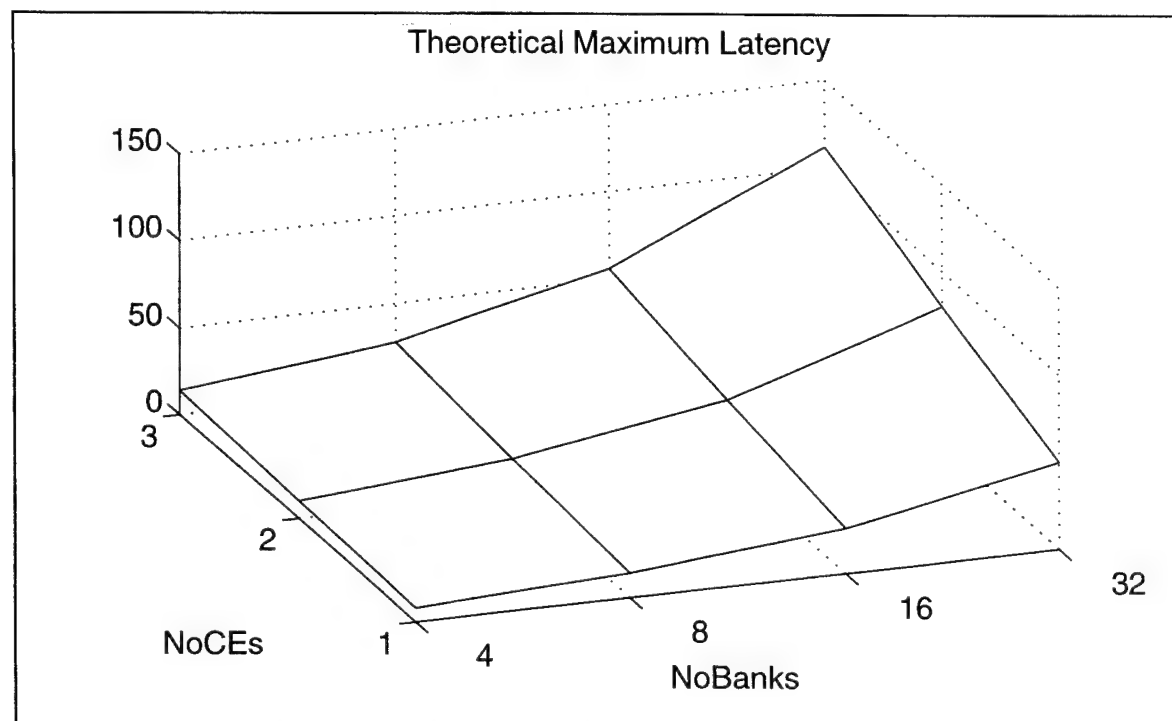
**Figure VI.1 Steady-State Throughput for Strides=1,3,5,7,9 (Conventional Decoding)**



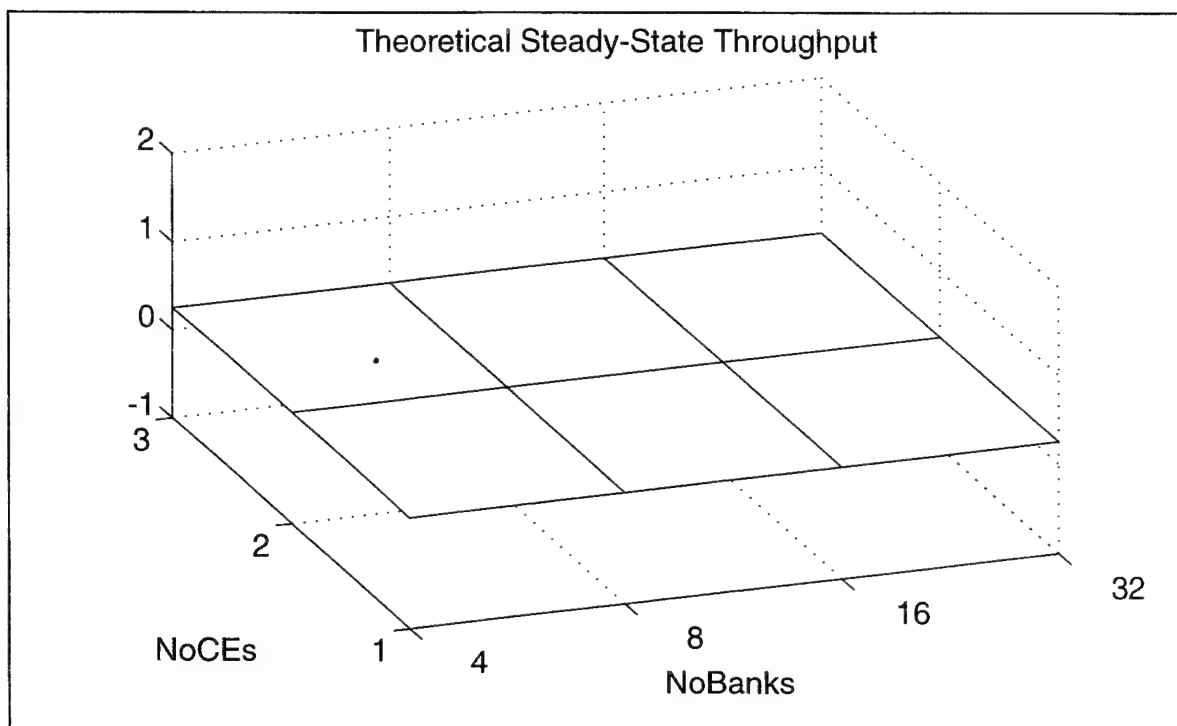
**Figure VI.2 Maximum Latency for Strides=1,3,5,7,9 (Conventional Decoding)**



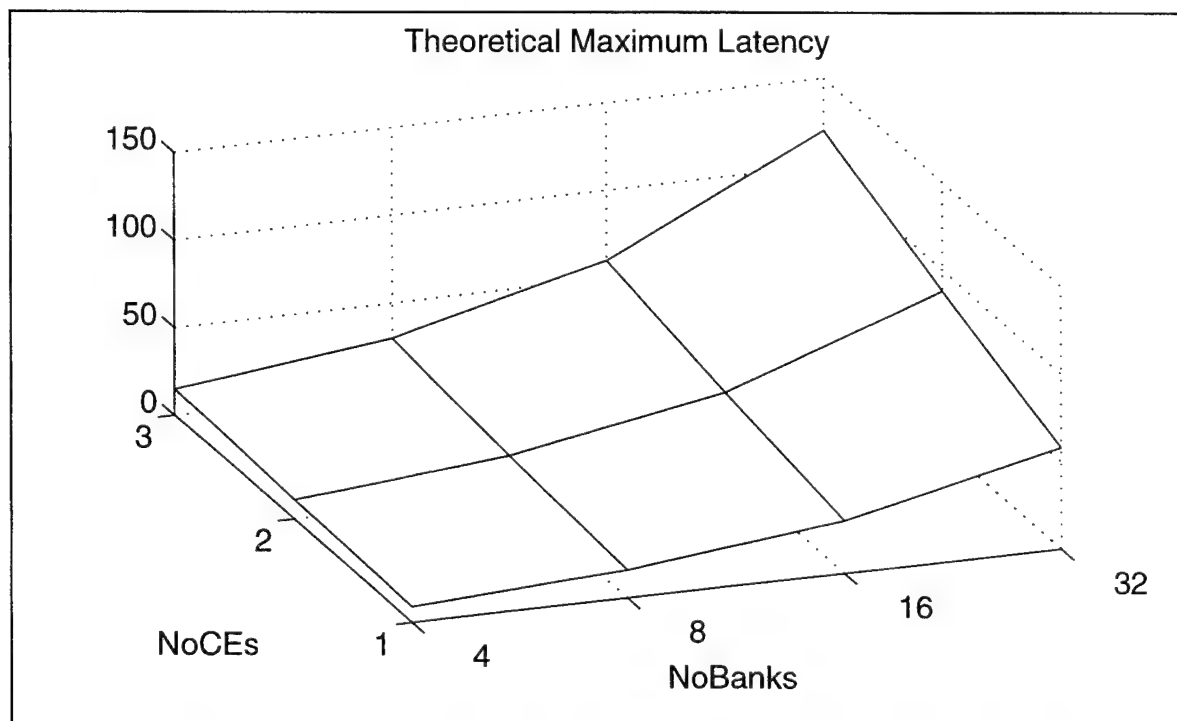
**Figure VI.3 Steady-State Throughput for Stride=2, 6 (Conventional Decoding)**



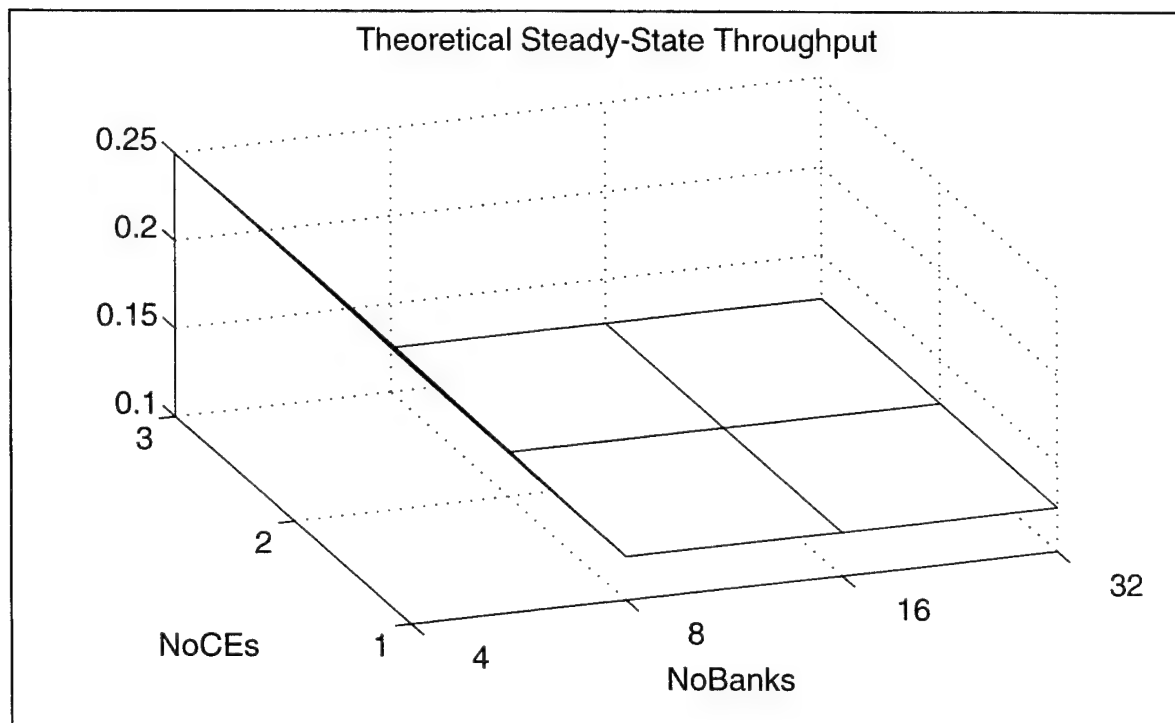
**Figure VI.4 Maximum Latency for Stride=2, 6 (Conventional Decoding)**



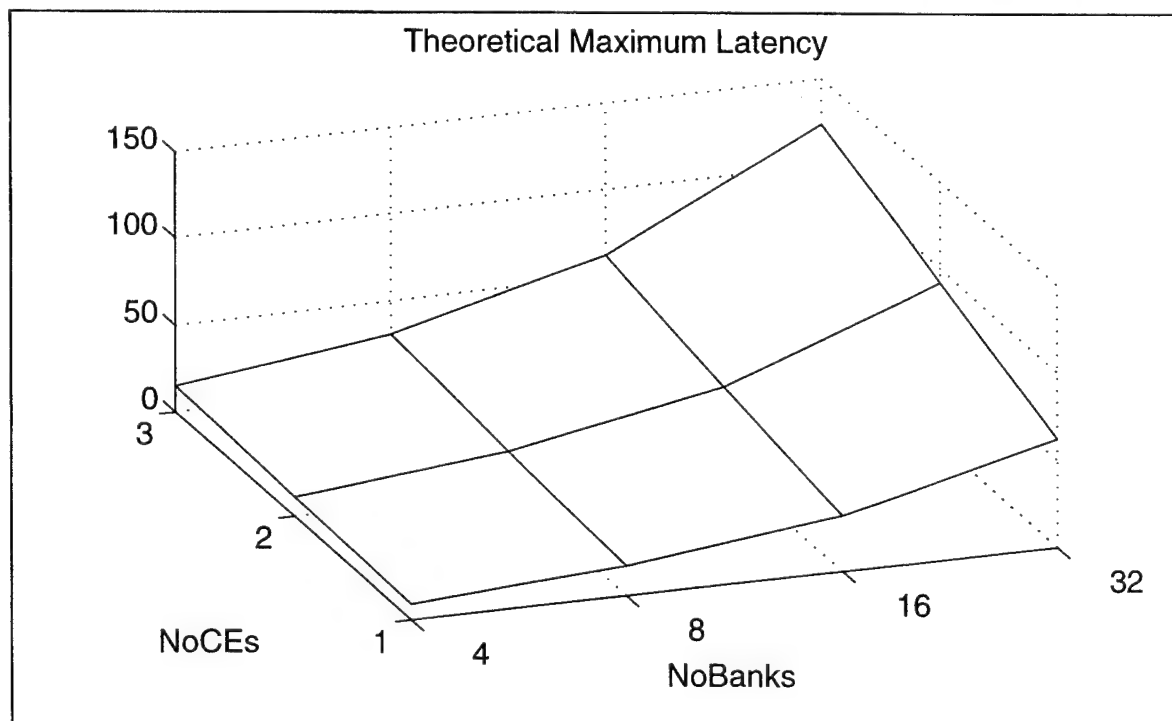
**Figure VI.5 Steady-State Throughput for Stride=4 (Conventional Decoding)**



**Figure VI.6 Maximum Latency for Stride=4 (Conventional Decoding)**



**Figure VI.7 Steady-State Throughput for Stride=8 (Conventional Decoding)**



**Figure VI.8 Maximum Latency for Stride=8 (Conventional Decoding)**

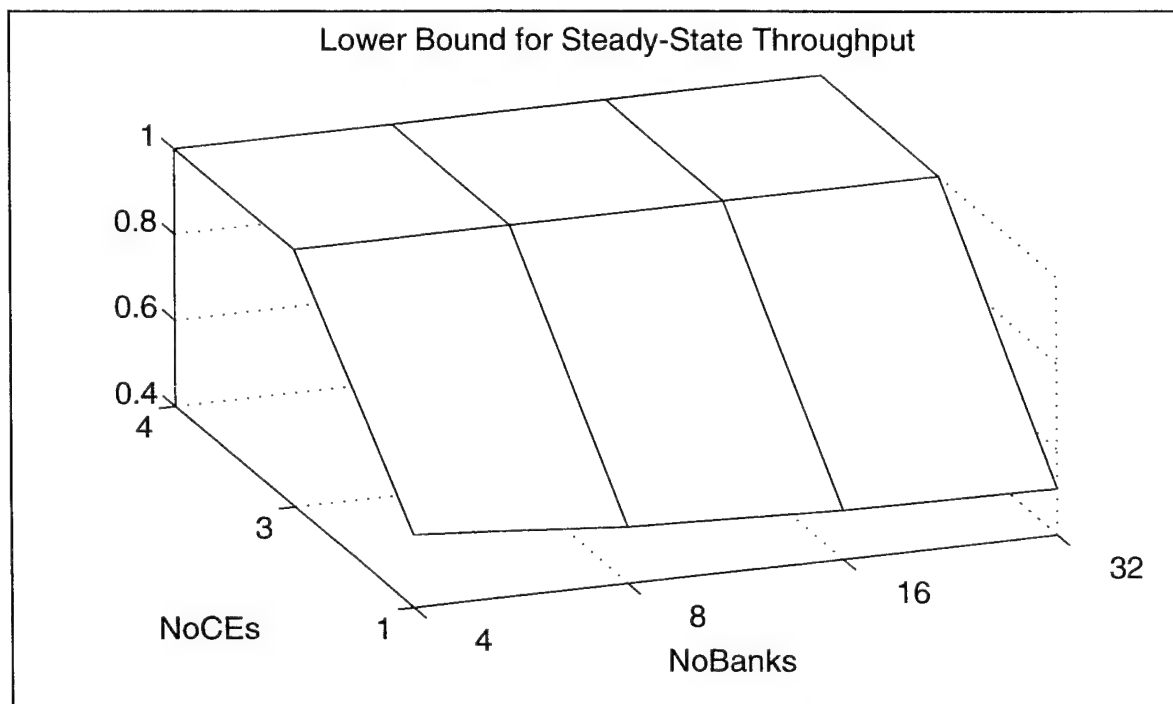
The second experiment is designed to validate the effectiveness of permutation-based techniques when applied to constant-stride address streams. Further, the effects on latency will be examined carefully because the latency analysis only provides an expression for an upper bound. The parameter values for the number of banks and the memory ratio is the same as in the previous experiment. The values used for the number of cache elements:

$$\text{NoCE} = 1, 3, 4. \quad (\text{VI.2})$$

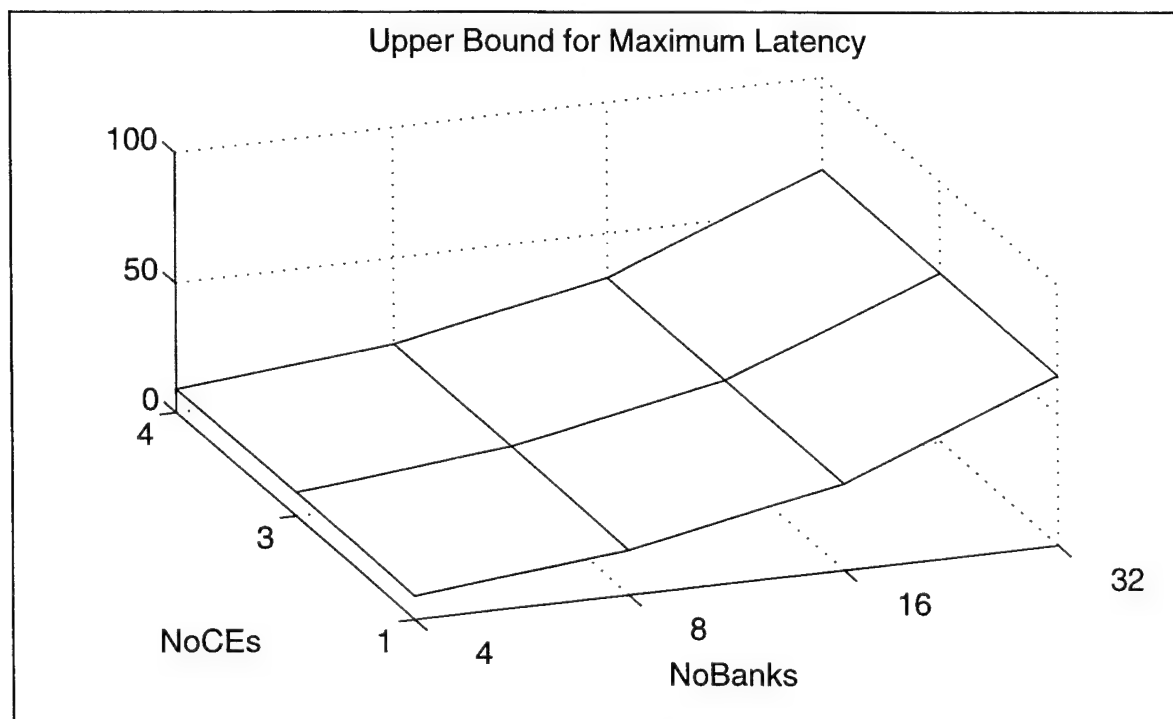
The first value provides for the standard interleaving case. A value of three is the value required for optimal steady-state throughput. The value of four is added for sensitivity analysis.

This experiment is designed to validate expressions for the steady-state throughput and latency as expressed in Equations (VI.24) through (VI.27). Plots generated based on these equations are shown in Figure VI.9 and Figure VI.10. Figure VI.9 illustrates the steady-state throughput of unity for all strides that are a power of two when permutation-based decoding is used. The corresponding upper bound of the maximum latency is shown in Figure VI.10. Note that no theoretical results exist for strides that are not a power of two for permutation-based memory decoding. Two strides (stride=3 and 5) are simulated to provide exemplar performance parameters when the stride is not a power of two. Although it is desirable for all strides to yield optimal performance, those strides which are not powers of two are not required for the vector architecture described in Chapter 0.





**Figure VI.9 Steady-State Throughput for Stride= $2^k$  for  $k = 0, 1, 2 \dots$  (Permutation-Based Decoding)**



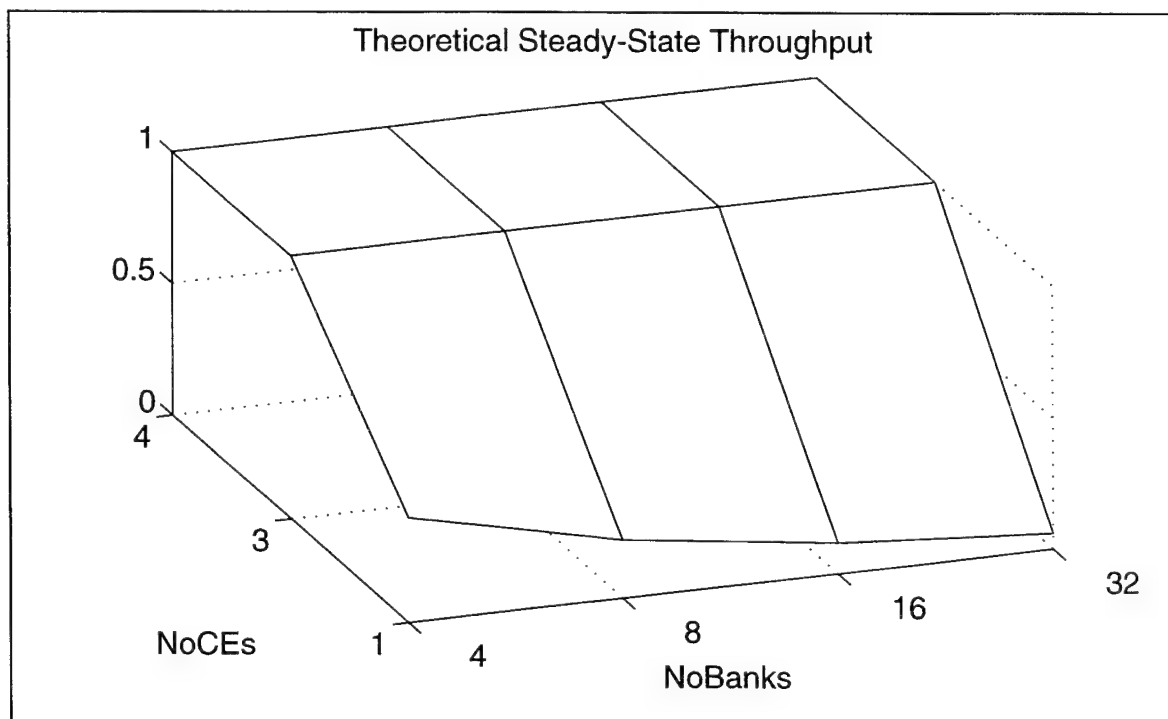
**Figure VI.10 Maximum Latency for Stride= $2^k$  for  $k = 0, 1, 2 \dots$  (Permutation-Based Decoding)**

The third experiment is designed to verify the steady-state throughput and maximum latency for radix- $r$  butterfly address patterns when conventional decoding is used. Expressions for steady-state throughput and maximum latency are found in Equations (VI.12), (VI.13), (VI.15), and (VI.16). The number of banks and memory ratio parameters are identical to those used above. The values used for the number of cache elements are adjusted for each radix. In general, the number of cache elements must be equal to  $r+1$  where  $r$  is the radix value. The values used for each radix is indicated in Table VI.2. As in the previous experiments, the standard interleaving case is included as well as the value that the analysis indicates will provide optimum steady-state throughput.

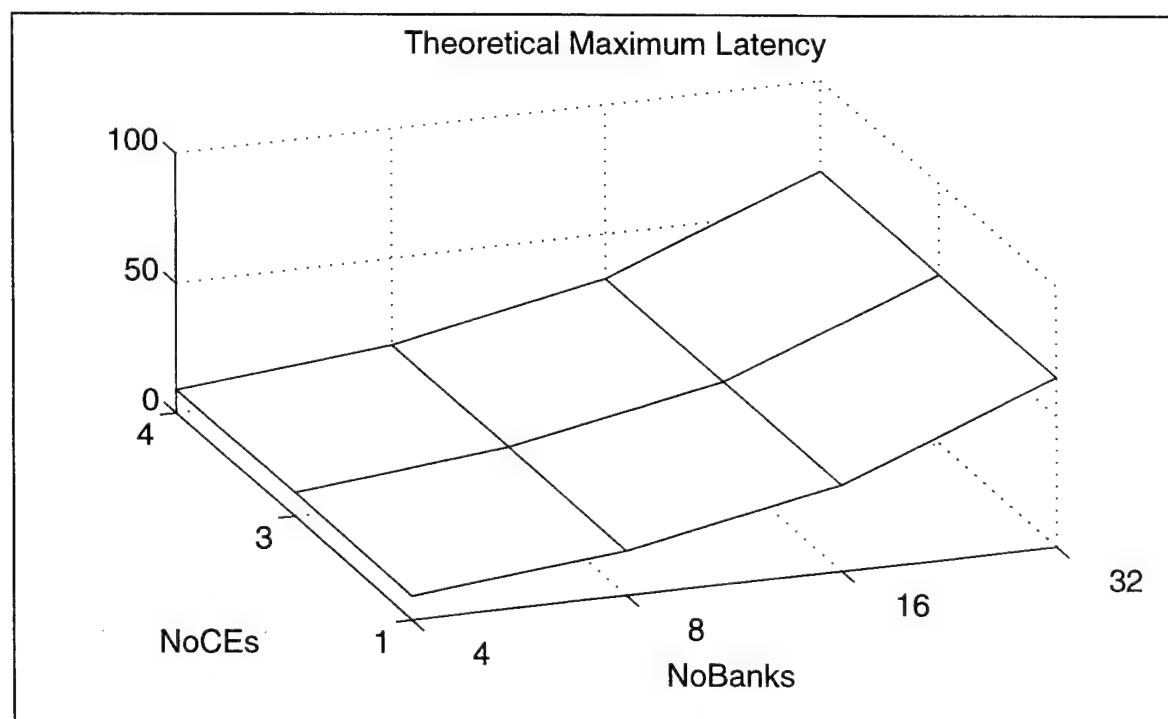
Radix	NoCE Evaluated
2	1, 3, 4
4	1, 5, 6
8	1, 9, 10
16	1, 17, 18

**Table VI.2 NoCE Evaluated in the Third Vector Processor Experiment**

The plots for these theoretical results are shown in Figure VI.11 through Figure VI.15. Figure VI.11 illustrates the theoretical steady-state throughput for radix-2 butterfly address patterns for conventional decoding. Note that for the standard interleaving cases, the value represents a lower bound. The remaining steady-state throughput plots (radix-4, 8, and 16) are not shown because the variation in these plots is within four percent of that shown in Figure VI.11 and that variation occurs only for the standard interleaving cases. The upper bound for the maximum latencies are shown in Figure VI.12 through Figure VI.15 for radices of two, four, eight, and 16 respectively. Although the basic shape of these plots are similar, the scale is seen to increase as the value of the radix increases.



**Figure VI.11 Steady-State Throughput for Radix=2 (Conventional Decoding)**



**Figure VI.12 Maximum Latency for Radix=2 (Conventional Decoding)**

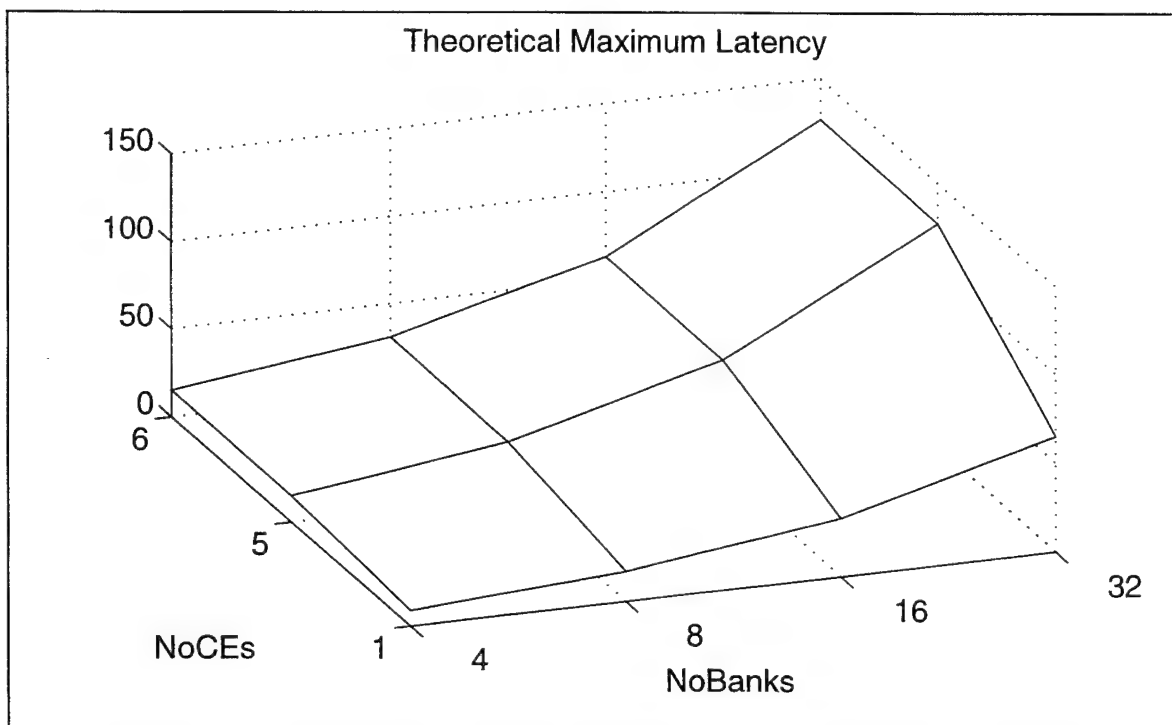


Figure VI.13 Maximum Latency for Radix=4 (Conventional Decoding)

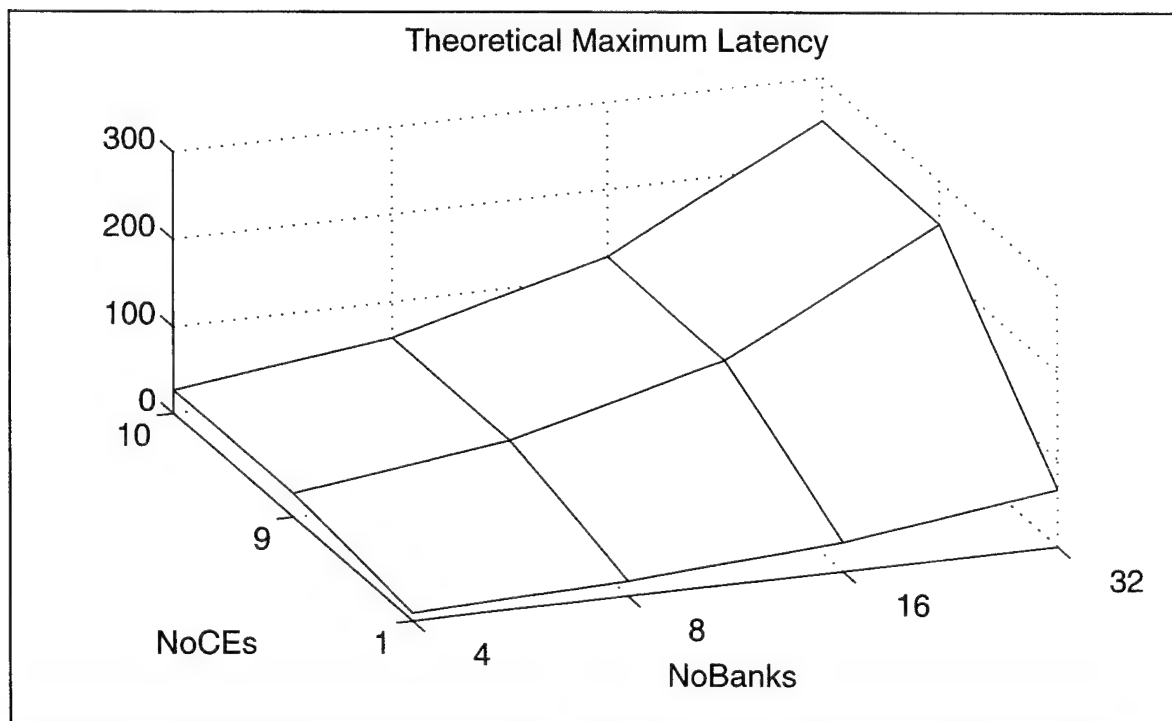
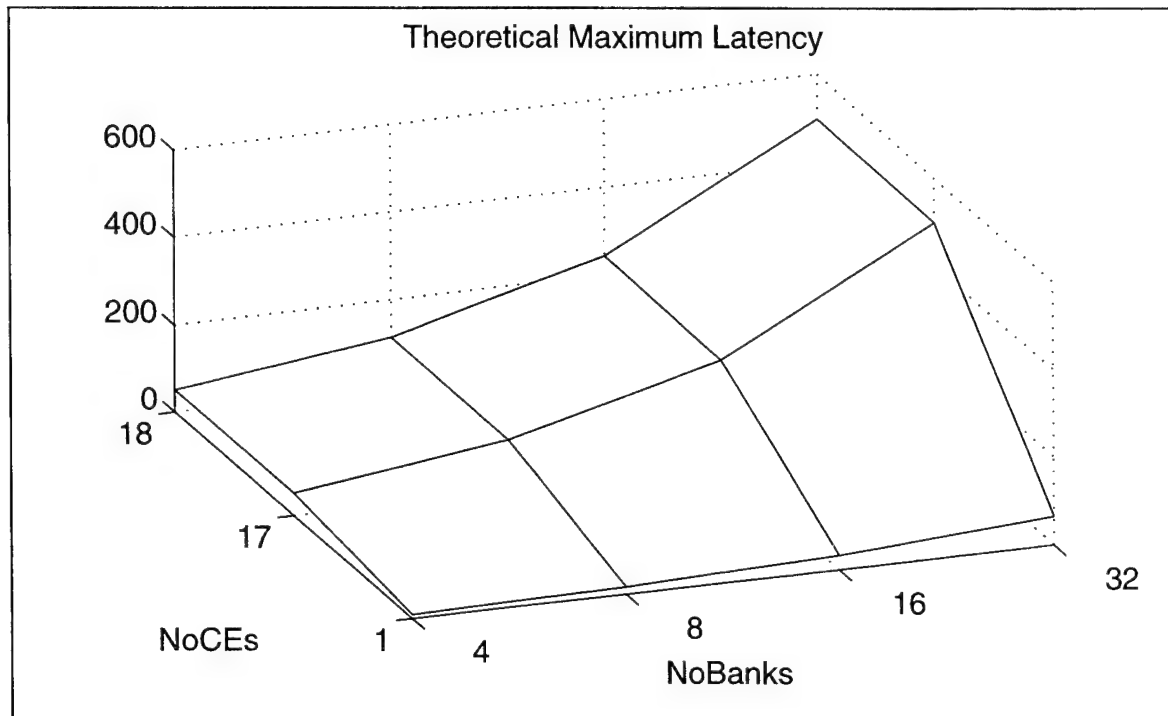


Figure VI.14 Maximum Latency for Radix=8 (Conventional Decoding)



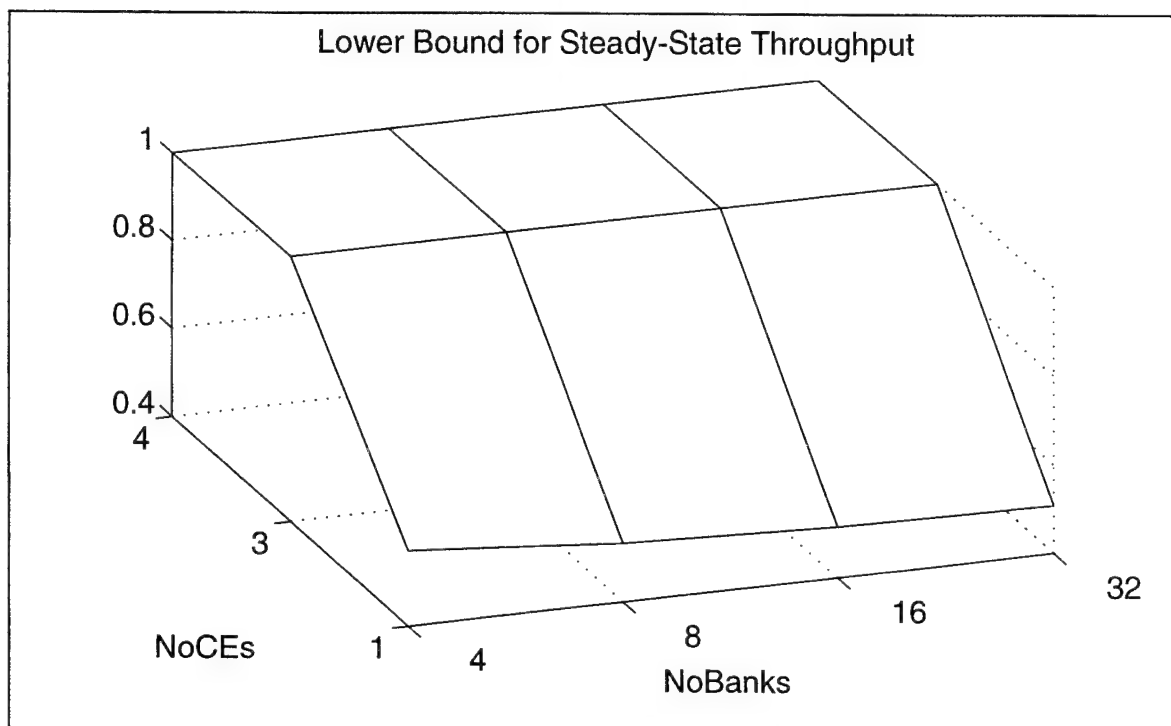
**Figure VI.15 Maximum Latency for Radix=16 (Conventional Decoding)**

The fourth vector processor experiment is similar to the previous experiment except that permutation-based decoding is used and the values selected for the number of cache elements are adjusted to yield optimum steady-state throughput for radix- $r$  address patterns for tailored permutation-based memory encoding. The values used for the number of cache elements are

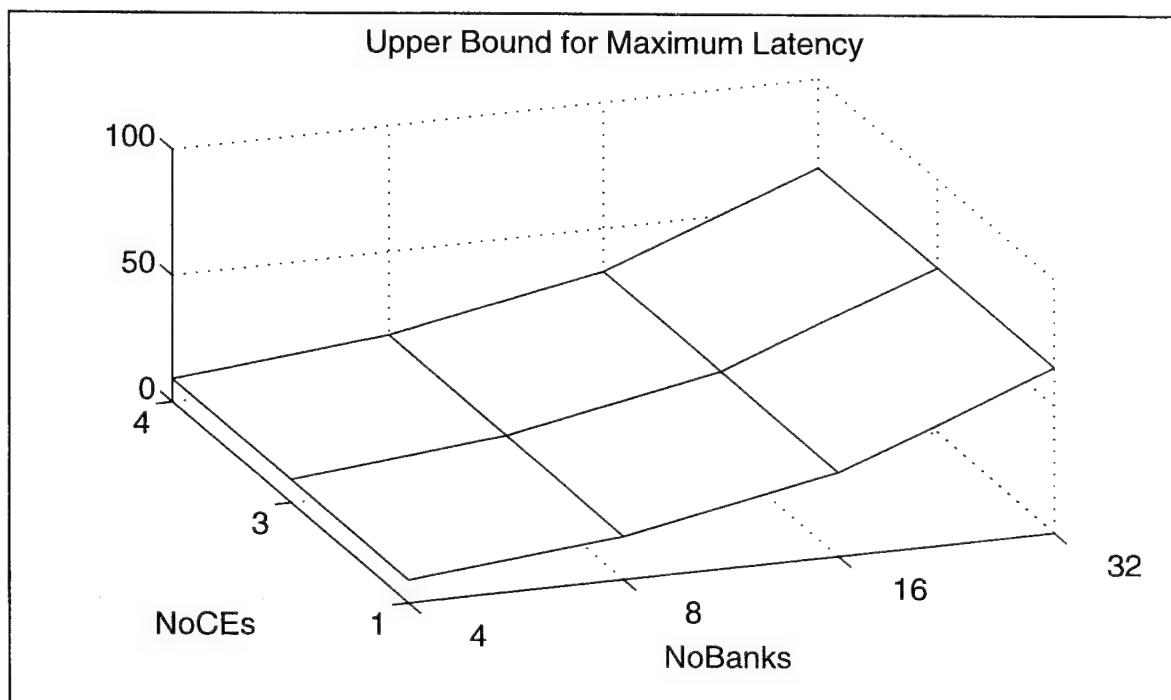
$$\text{NoCE} = 1, 2, 3, 4$$

for all radices. Pertinent performance expressions are found in Equations (VI.24) through (VI.27). These are the equations for constant stride but are also appropriate for radix- $r$  butterfly patterns when the specialized matrices are used.

The plots for these theoretical results are shown in Figure VI.16 and Figure VI.17. Figure VI.16 illustrates the theoretical lower bound for steady-state throughput for radix- $r$  butterfly address patterns for all radices for permutation-based decoding. The maximum latency plot for all radices is shown in Figure VI.17. Observe that the maximum latency for this case is anticipated to be substantially lower than for the conventional decoding for the higher radices.



**Figure VI.16 Steady-State Throughput for Radix=2, 4, 8, and 16 (Permutation-Based Decoding)**



**Figure VI.17 Maximum Latency for Radix=2, 4, 8, and 16 (Permutation-Based Decoding)**

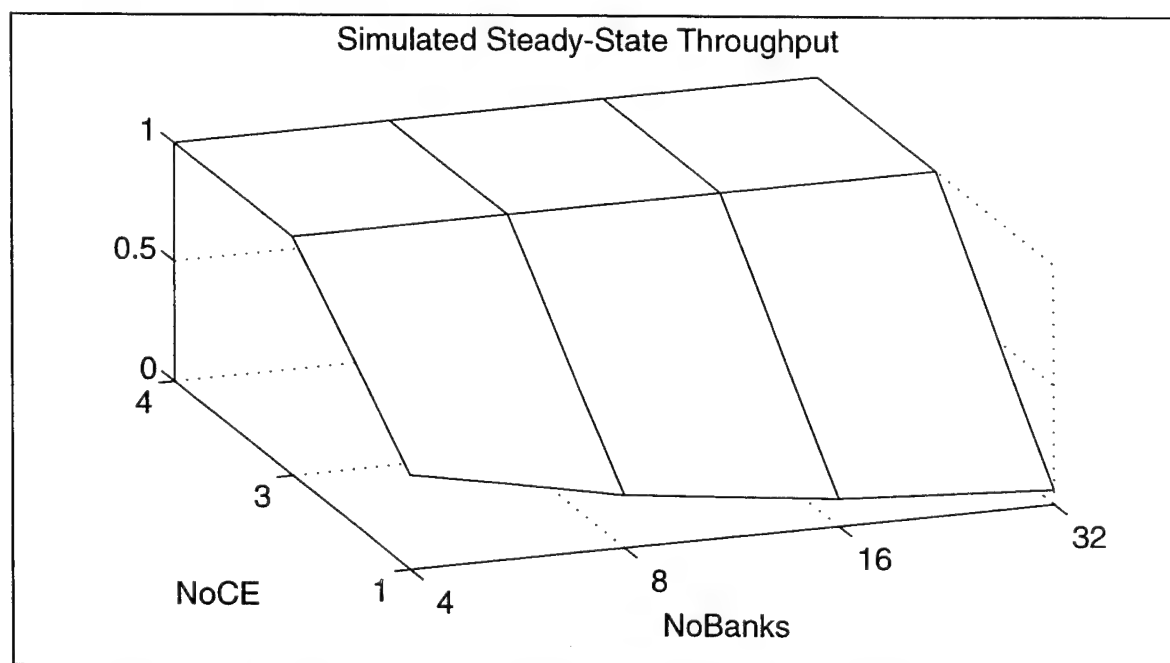
The fifth and sixth vector processor experiments use digit-reversed address patterns. The fifth experiment uses conventional decoding. Digit-reversed address patterns are characterized by  $r$ -length sequences of constant stride  $r^{NoDigits-1}$  where  $r$  is the radix of the FFT and  $NoDigits$  is the number of digits required to represent the address of the vector. Since the effective number of banks is governed by Equation (V.1) the effective number of banks is always one when  $r$  and the number of banks are both a power of two. The following data set is used to validate this result:

$$NoBanks = 4, 8, 16, 32$$

$$MemRatio = NoBanks$$

$$NoCE = 1, 3, 4.$$

The theoretical steady-state throughput is illustrated in Figure VI.18. Observe that the steady-state throughput is inverse of the number of banks and this value is invariant to the number of cache elements.



**Figure VI.18 Steady-State Throughput for Radix=2 (Conventional Decoding)**

The sixth experiment evaluates several digit-reversed patterns using permutation-based memory decoding. Expressions for steady-state throughput and maximum latency are found in Equations (VI.31) and (VI.32), as well as (VI.24) through (VI.27). The

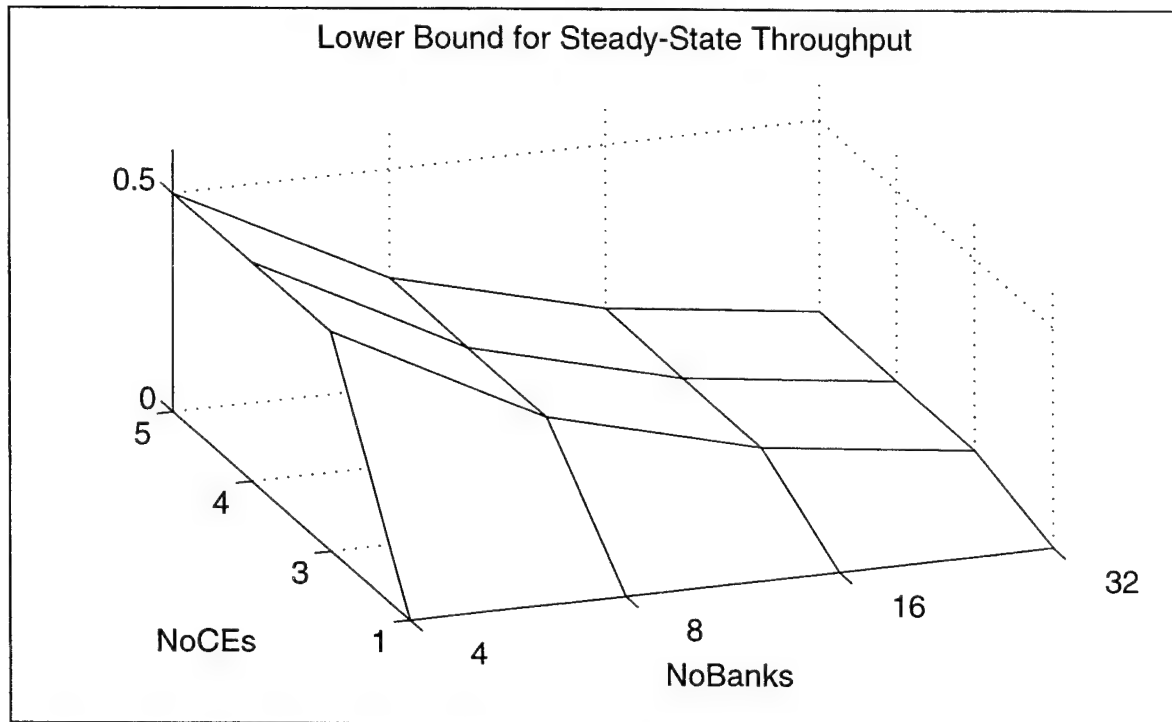
number of banks and memory ratio parameters are identical to those used above. The values used for the number of cache elements are shown in Table VI.3. In general, the number of cache elements used in this experiment is the same as those used in the second experiment. Note that the value of five was added for radix eight and sixteen after one iteration of simulations. This will be discussed further in the next section.

Radix/NoDigits	NoCE Evaluated
2/10	1, 3, 4
4/5	1, 3, 4
8/4	1, 3, 4, 5
16/3	1, 3, 4, 5

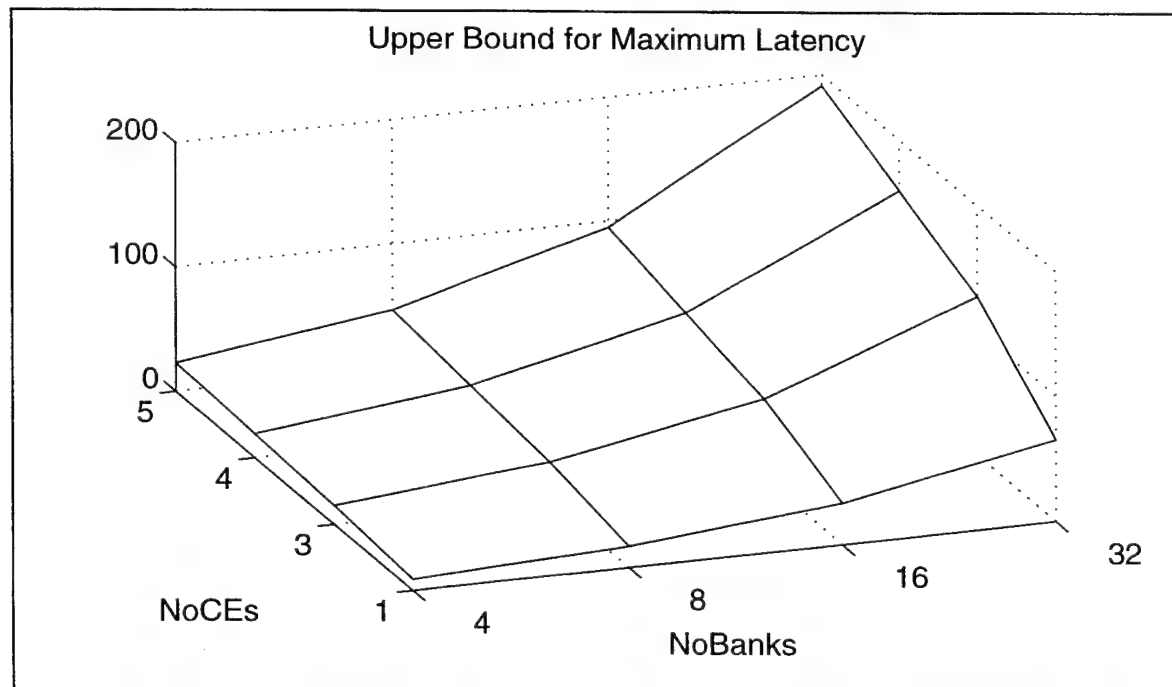
**Table VI.3 NoCE Evaluated in the Sixth Vector Processor Experiment**

The theoretical results for the steady-state throughput and maximum latency for the four cases shown in Table VI.3 are shown in Figure VI.19 through Figure VI.26. The steady-state throughput plots are lower bounds except when the throughput is optimum. The lower bound occurs whenever the base is less than the number of banks as described in Chapter V Section D.

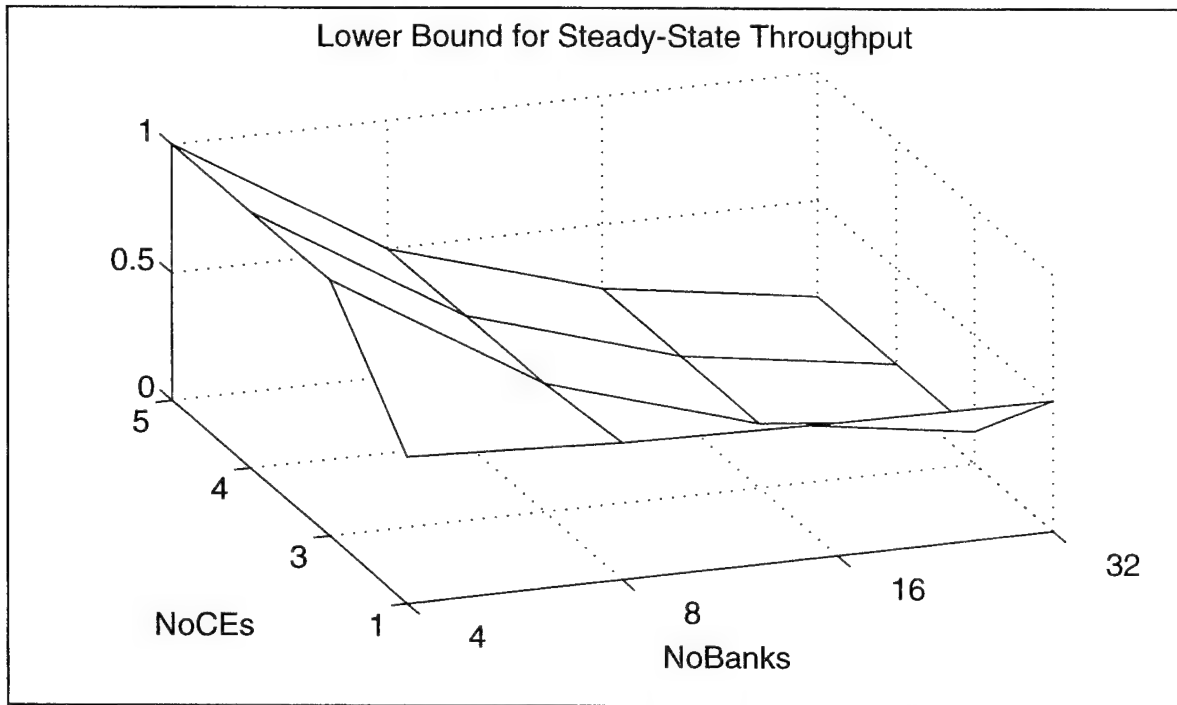




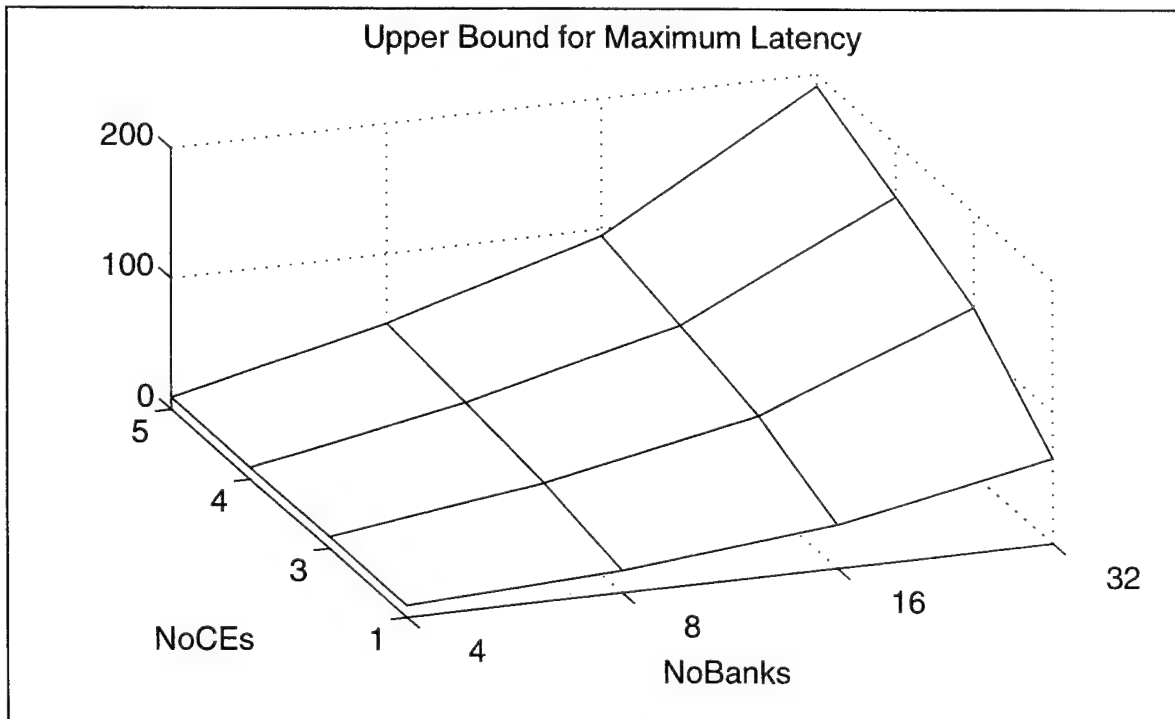
**Figure VI.19 Steady-State Throughput for Radix=2/NoDigits=10 (Permutation-Based Decoding)**



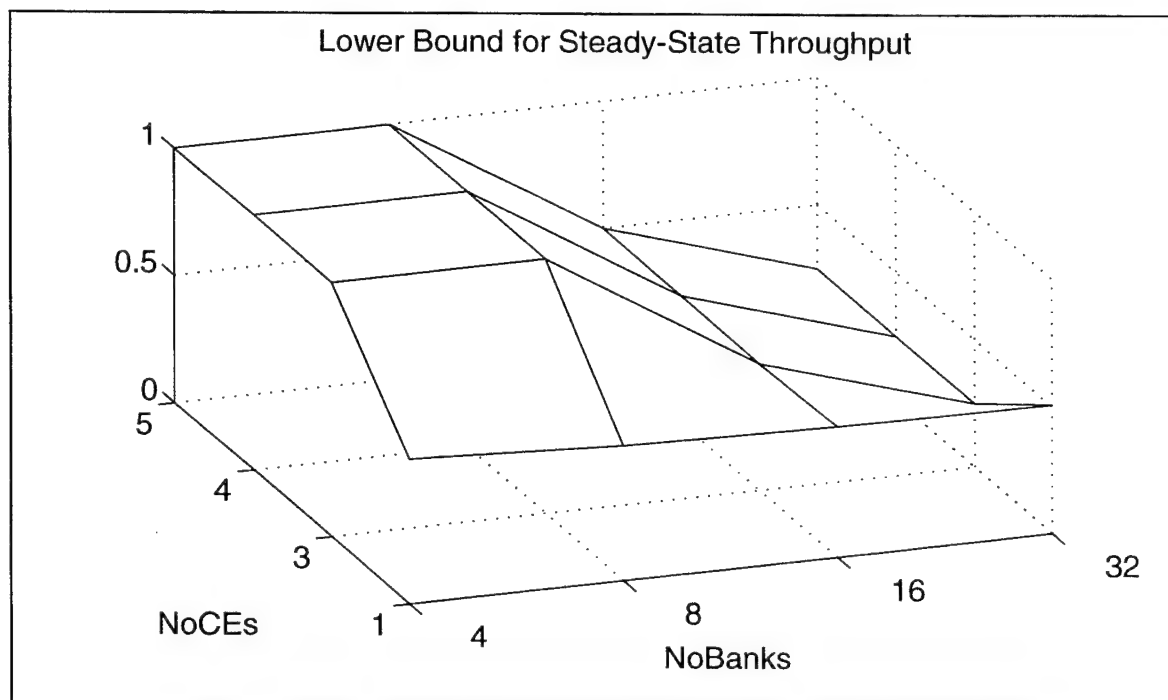
**Figure VI.20 Maximum Latency for Radix=2/NoDigits=10 (Permutation-Based Decoding)**



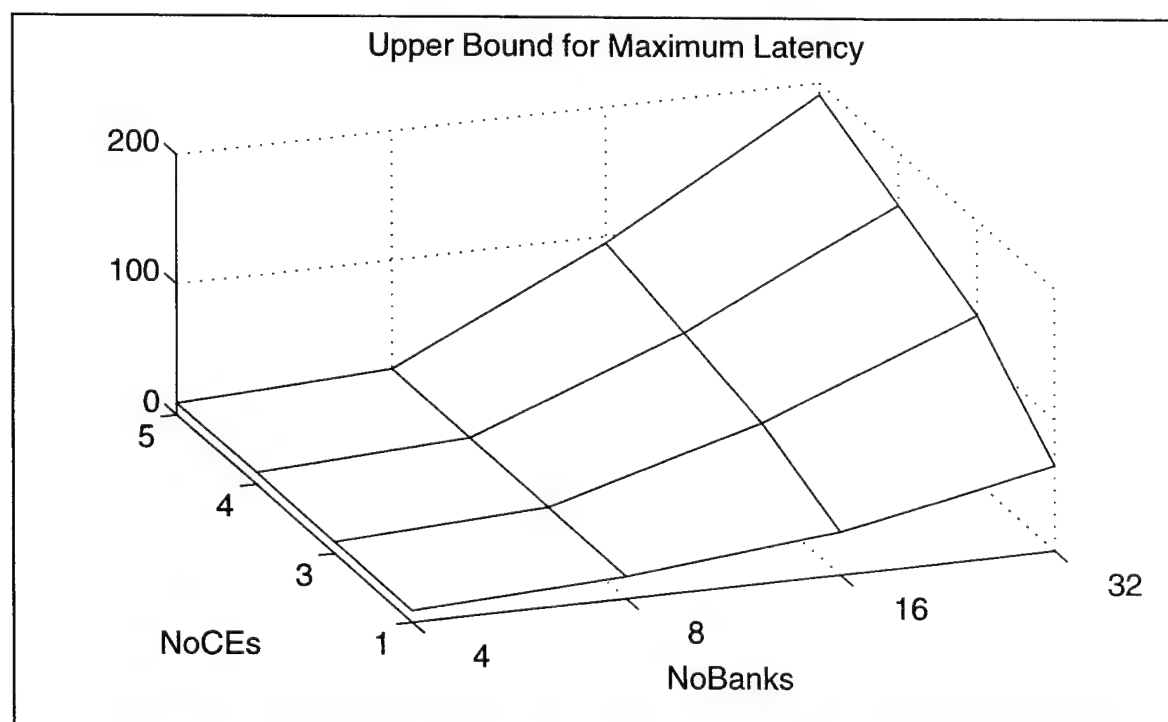
**Figure VI.21 Steady-State Throughput for Radix=4/NoDigits=5 (Permutation-Based Decoding)**



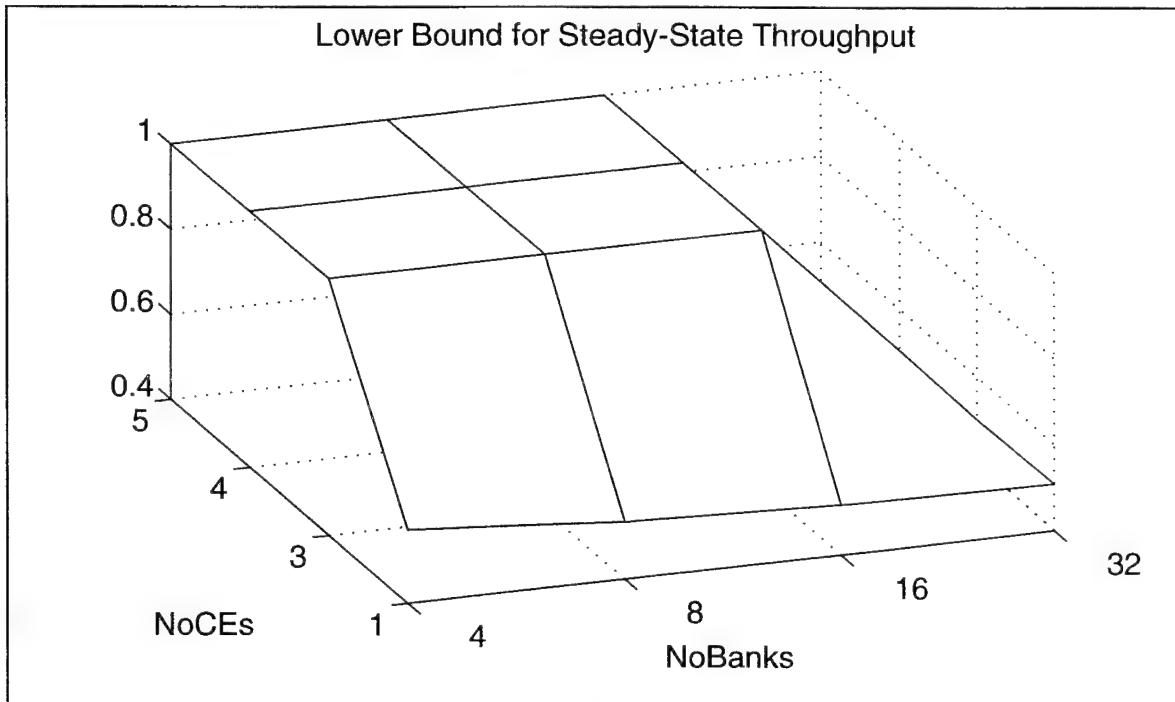
**Figure VI.22 Maximum Latency for Radix=4/NoDigits=5 (Permutation-Based Decoding)**



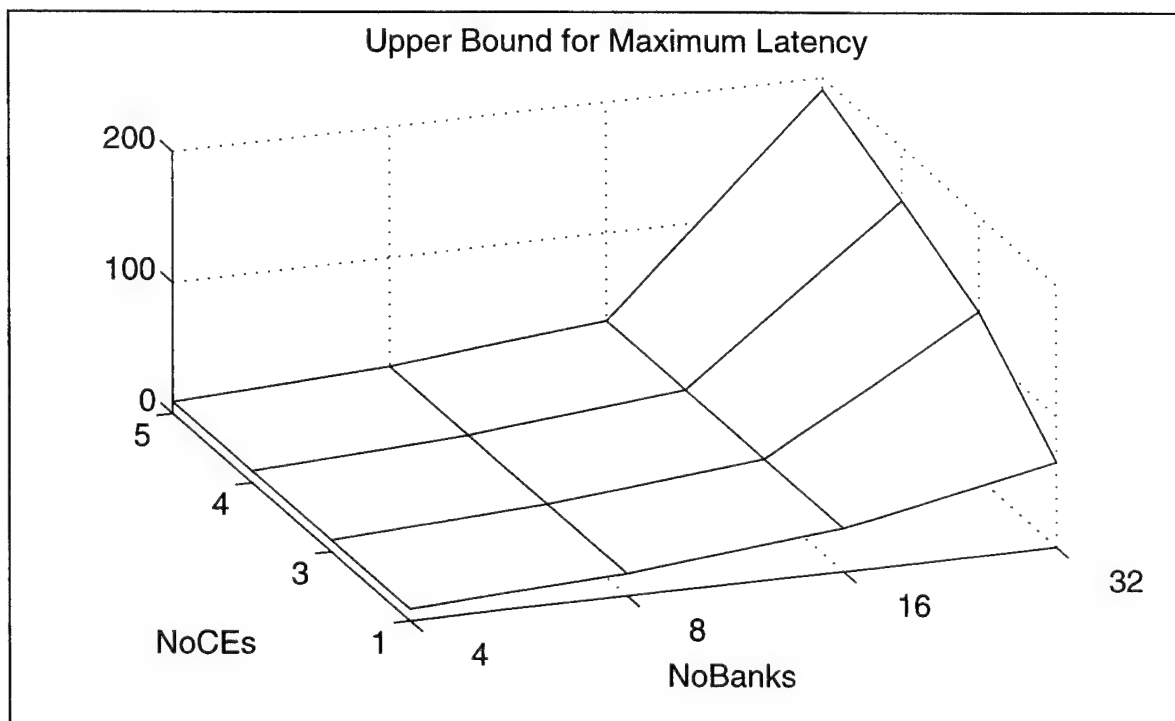
**Figure VI.23 Steady-State Throughput for Radix=8/NoDigits=4 (Permutation-Based Decoding)**



**Figure VI.24 Maximum Latency for Radix=8/NoDigits=4 (Permutation-Based Decoding)**



**Figure VI.25 Steady-State Throughput for Radix=16/NoDigits=3 (Permutation-Based Decoding)**



**Figure VI.26 Maximum Latency for Radix=16/NoDigits=3 (Permutation-Based Decoding)**

The second group of experiments pertain to general-purpose computing and are summarized in Table VI.4. This experiment examines the marginal effectiveness of adding additional memory banks and cache elements. The number of memory banks is matched to the memory ratio such that if the memory banks are used optimally, then the speedup obtained from the memory system is equal to the number of banks. The address stream is completely random to allow comparison with results in the literature [Ref 55]. The parameters used for this experiment are

$$\begin{aligned}
 \text{NoBanks} &= 1, 4, 8, 16, 32 \\
 \text{MemRatio} &= \text{NoBanks} \\
 \text{NoCE} &= 1, 2, 4, 8, 16, 32, 64 \\
 p &= 0.
 \end{aligned}
 \tag{VI.3}$$

Name	Purpose	Scope
Speedup Analysis	Investigate the affect to speedup when varying STM parameters. Set $p=0$ for historical comparison.	MemRatio == NoBanks for all cases.

**Table VI.4 General-Purpose Computer Experiment**

The next two sections contains the results of each of the simulation runs described above for vectoring processing and general-purpose computing respectively.

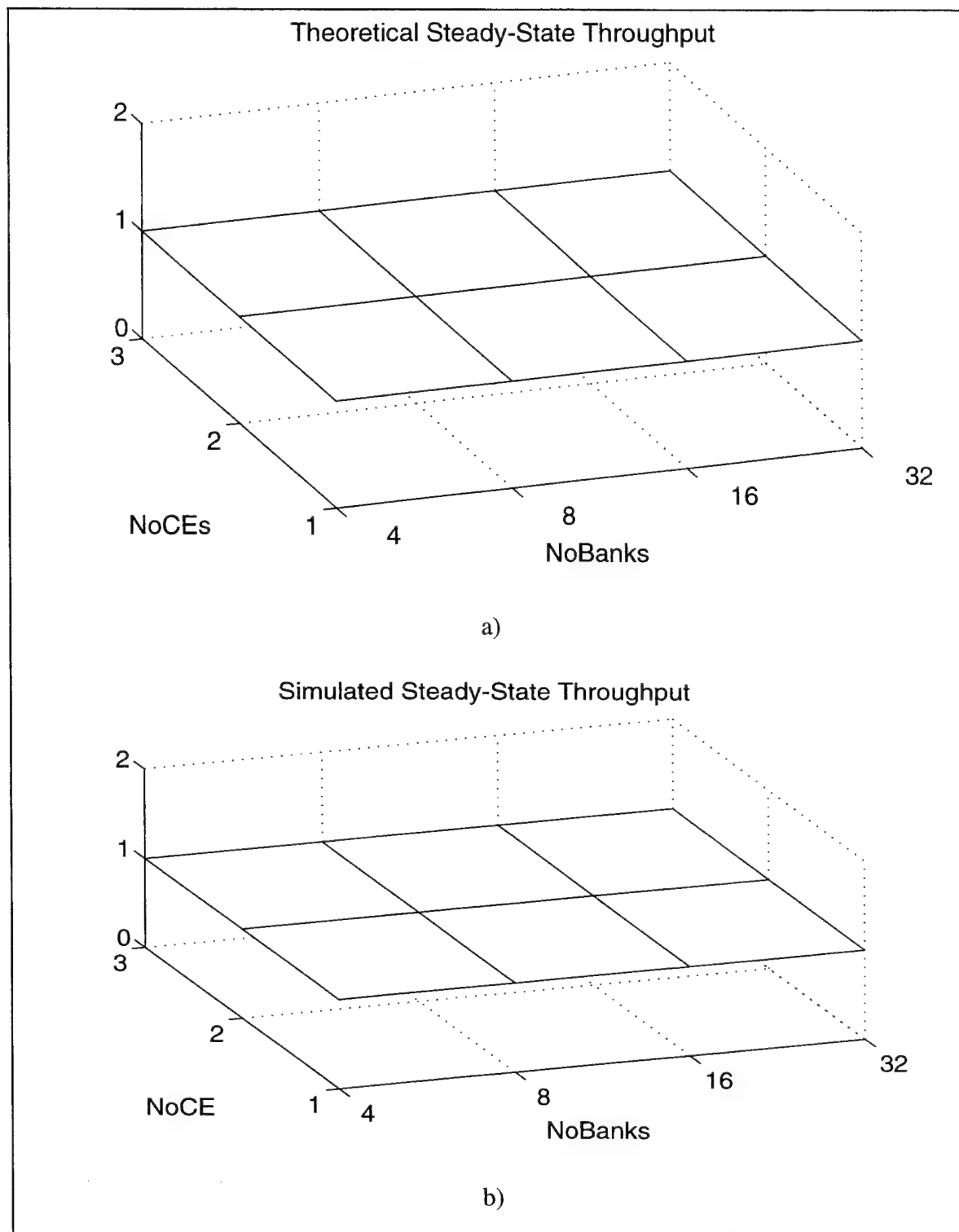
## **B. VECTOR PROCESSING EXPERIMENTS**

### **1. Constant Stride: Conventional Memory Decoding**

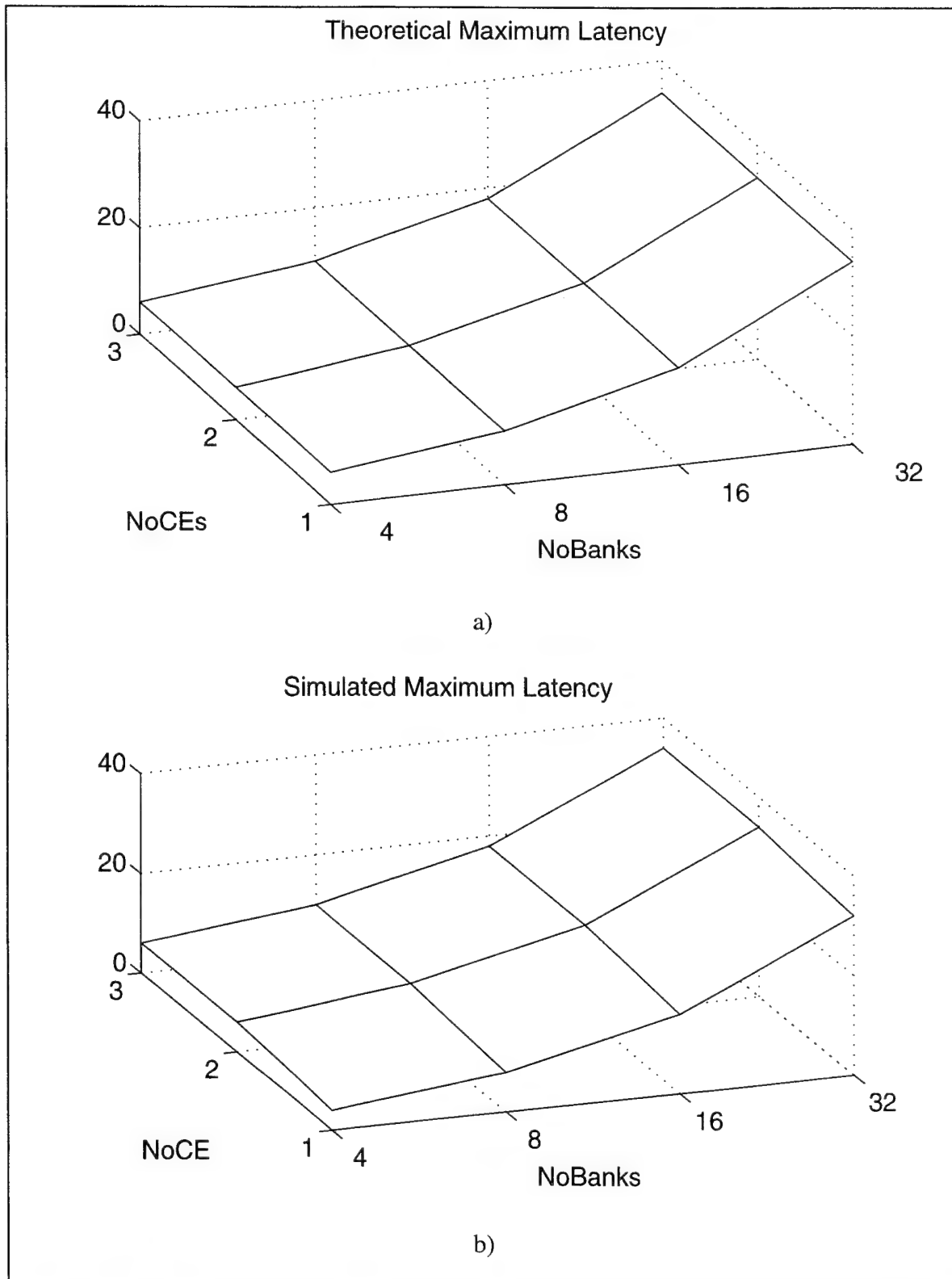
A comparison of the theoretical and simulated results for the first vector processor experiment are shown in Figure VI.27 through Figure VI.39. The plots for stride of one are shown in Figure VI.27 and Figure VI.28. For both of these performance measures, the theoretical and simulated results are identical.

Examples of two simulation runs, the first with four banks and two cache elements and the second with 32 banks and two cache elements are shown in Figure VI.29 and Figure VI.30 respectively. In each plot, the grant request line (GR) indicating memory requests are accepted by the memory, is active on the first cycle and remains active until all memory responses are accepted. The response enable (RE) line becomes

active indicating that output is available for the processor, and remains on until the last response is sent to the processor. In each case, the RE line follows the GR line after the necessary latency of six and 34, respectively (i.e.,  $MR+2$ ). This is the best performance that can be obtained from the memory systems. One of the tradeoffs of using a larger number of memory banks is the latency, as illustrated in Figure VI.29 and Figure VI.30. This latency results in a average throughput of 0.9624 and 0.795 for four versus 32 banks respectively. Figure VI.31 illustrates the effect on average throughput when varying the number of banks for the case of a stride of one. The penalty of a larger number of banks is clearly shown when the vectors are relatively small (128 points).



**Figure VI.27 Comparison of Theoretical Versus Simulated Steady-State Throughput for Strides=1,3,5,7,9 (Conventional Decoding)**



**Figure VI.28 Comparison of Theoretical Versus Simulated Maximum Latency for Strides=1,3,5,7,9 (Conventional Decoding)**



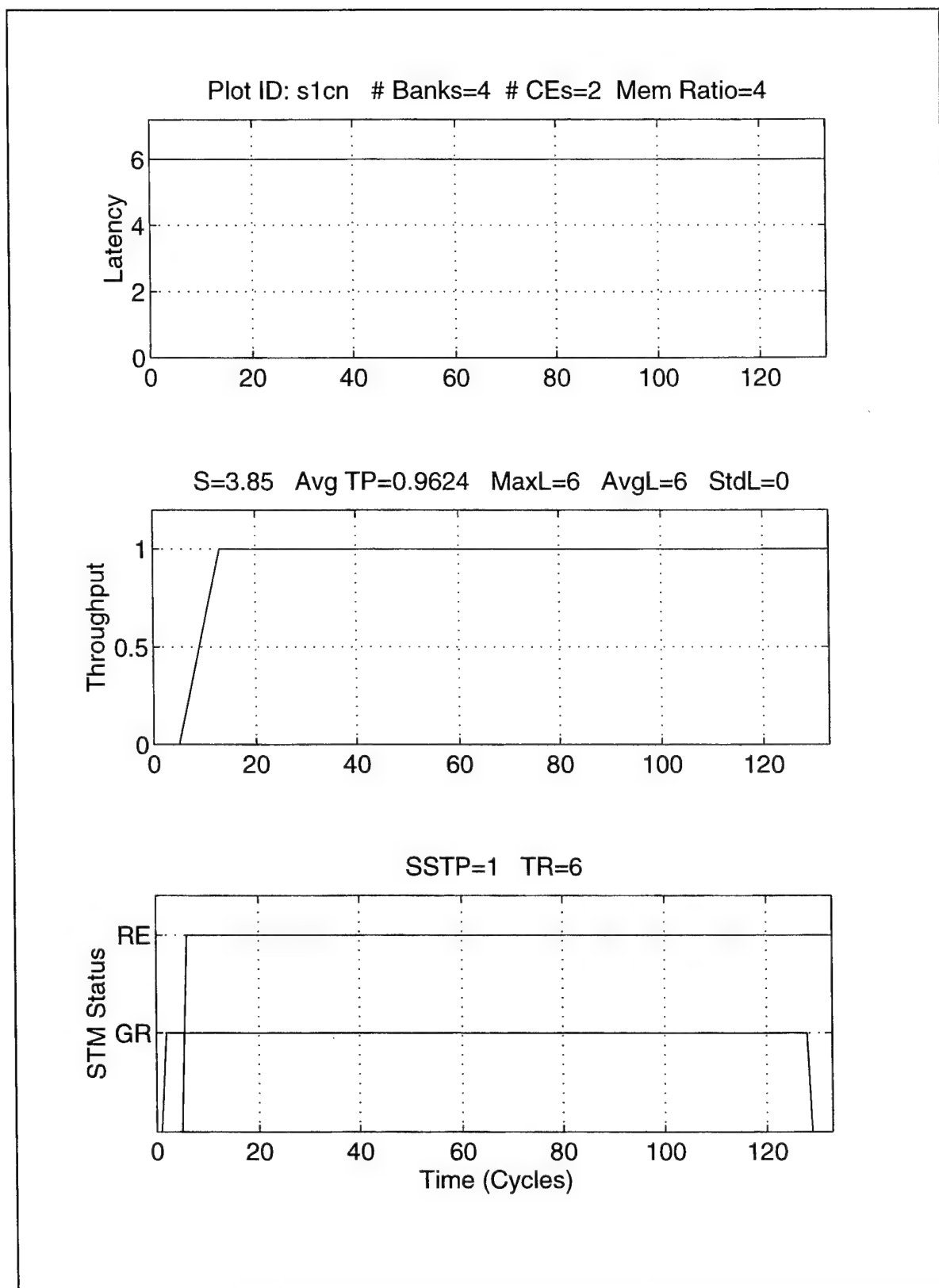


Figure VI.29 Detailed Simulation Run for Stride=1 STM(4,2,4)

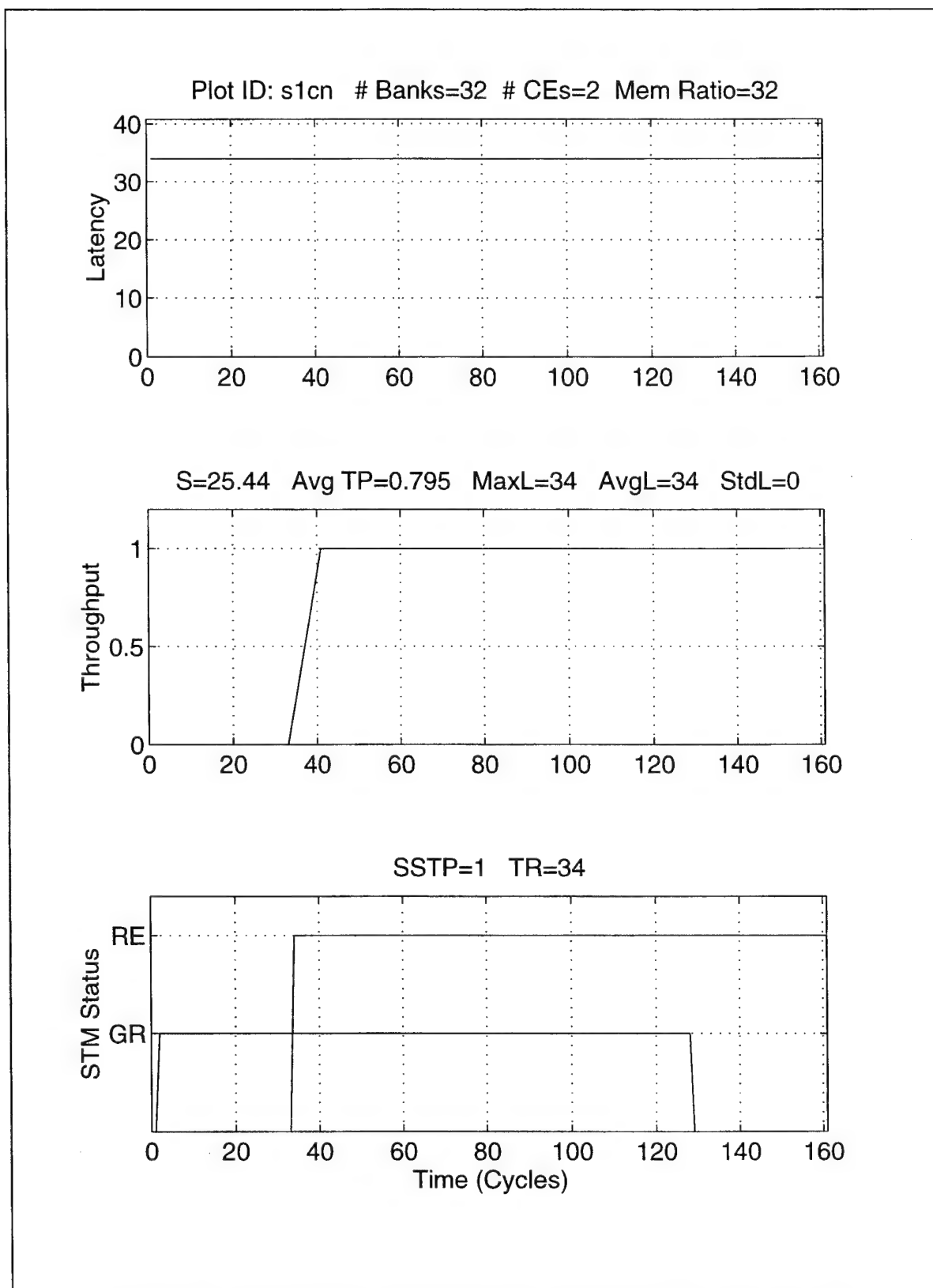
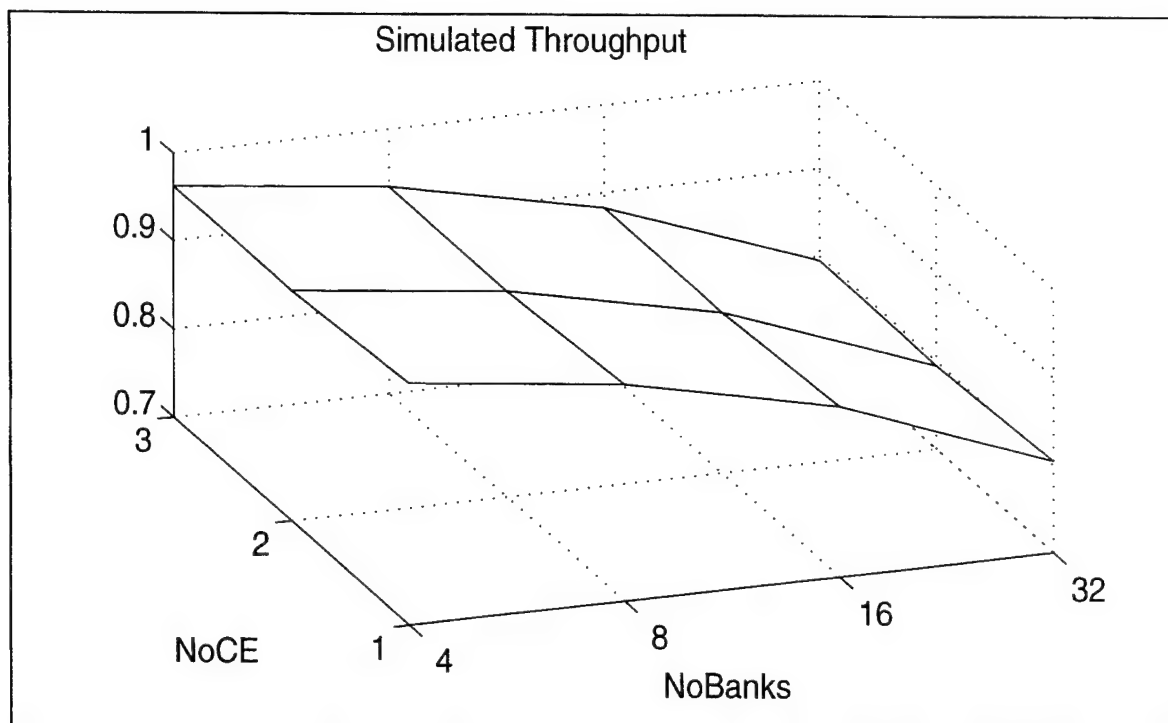


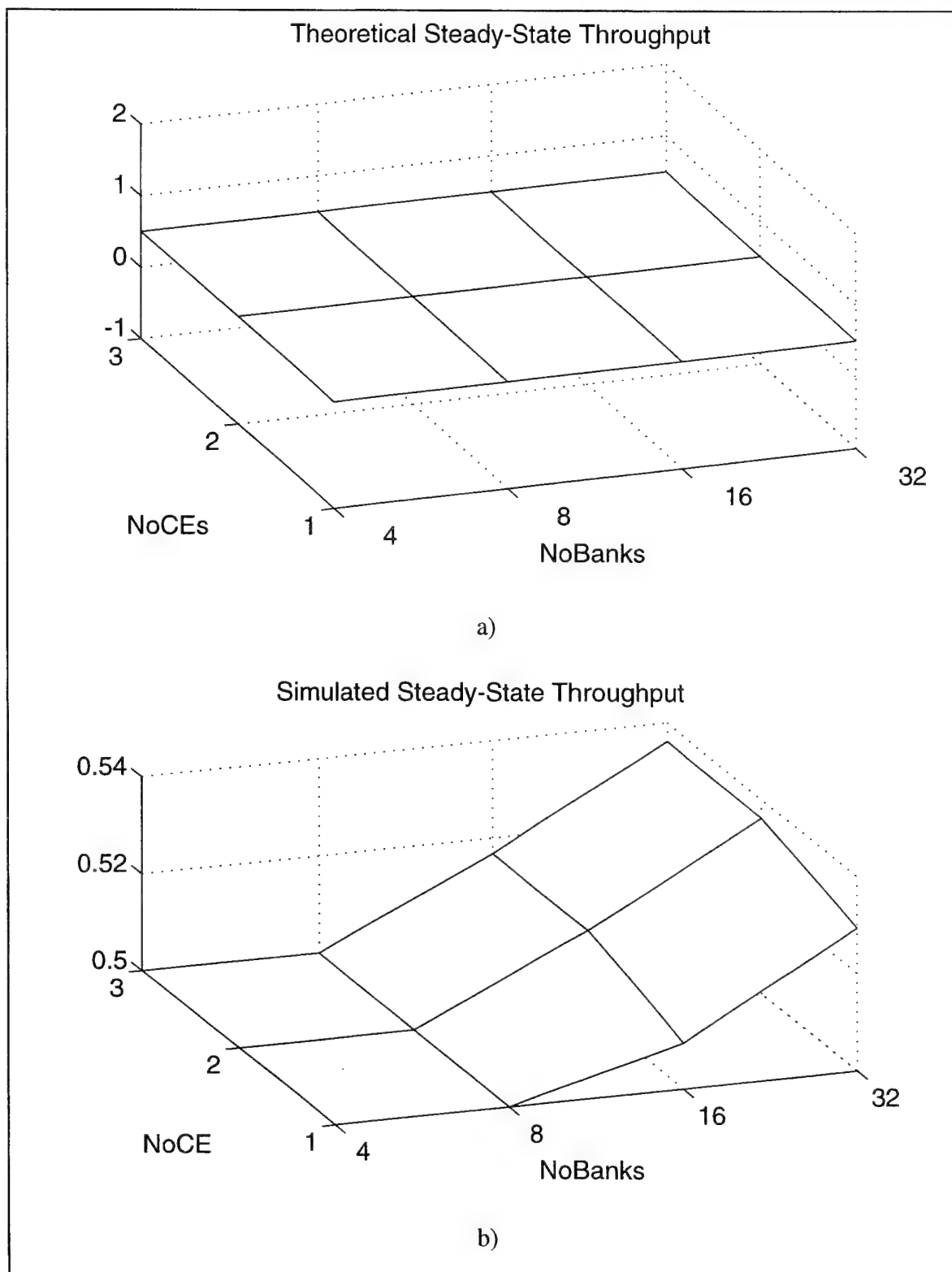
Figure VI.30 Detailed Simulation Run for Stride=1 STM(32,2,32)



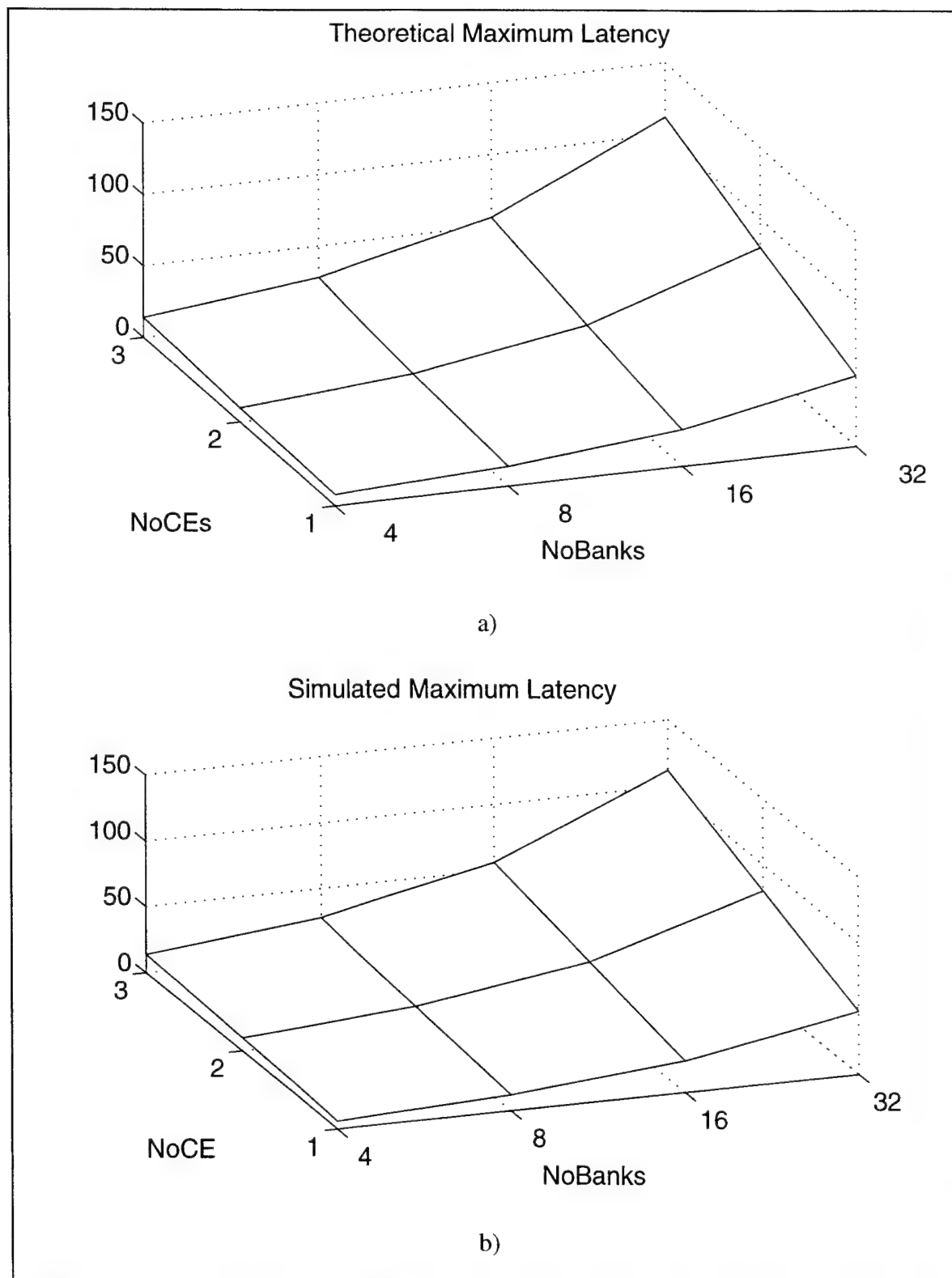
**Figure VI.31 Average Simulated Throughput for Stride=1 (Conventional Decoding)**

A comparison of the theoretical versus simulated steady-state throughput and maximum latency is shown in Figure VI.32 and Figure VI.33, respectively, for a stride of two. Notice that the simulated steady-state throughput varies by as much as four percent from the theoretical for thirty two banks. The steady-state throughput is calculated by taking the average of the last twenty five percent of the throughput values. This occasionally results in a bias error when the steady-state value of the throughput is not constant. Such a steady-state is illustrated in Figure VI.34 for thirty two banks and three cache elements.

The simulated maximum latency is in agreement with the theoretical plot for stride of two as shown in Figure VI.33. The average throughput, shown in Figure VI.35, indicates a consistent pattern with a stride of one. The average throughput obtained with four banks (approximately 0.495) is within one percent of the steady-state ceiling of 0.5.



**Figure VI.32 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=2 (Conventional Decoding)**



**Figure VI.33 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=2 (Conventional Decoding)**

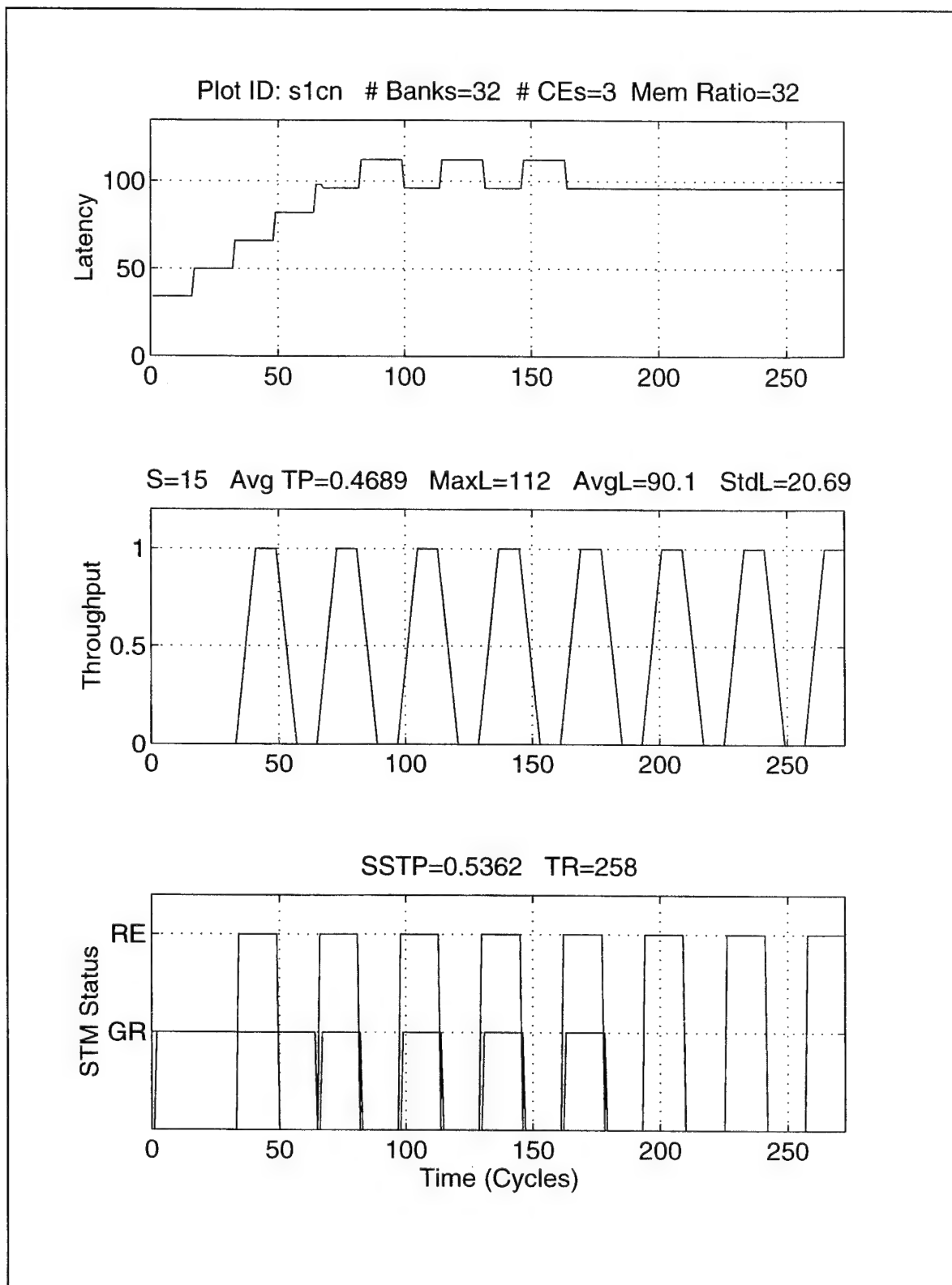
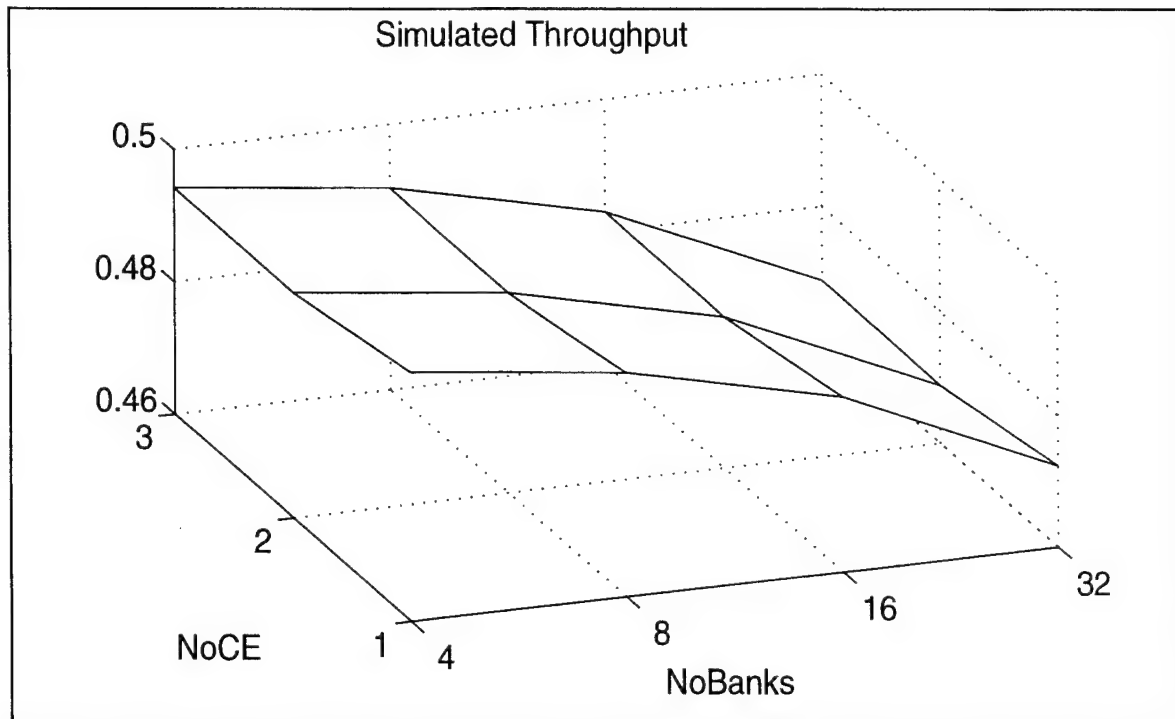


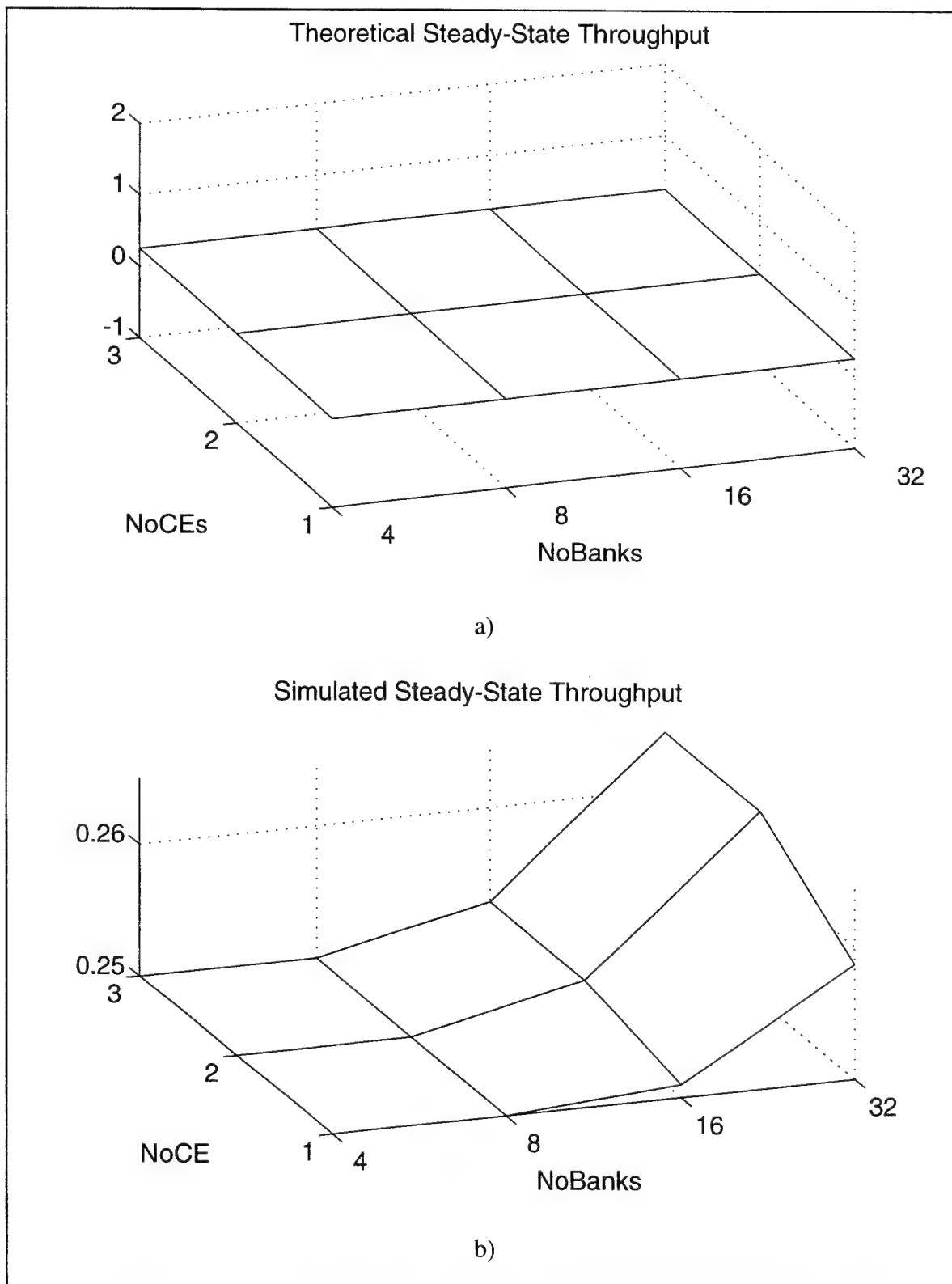
Figure VI.34 Detail Simulation Run for Stride=2 STM(32,3,32) (Conventional Decoding)



**Figure VI.35 Average Simulated Throughput for Stride=2 (Conventional Decoding)**

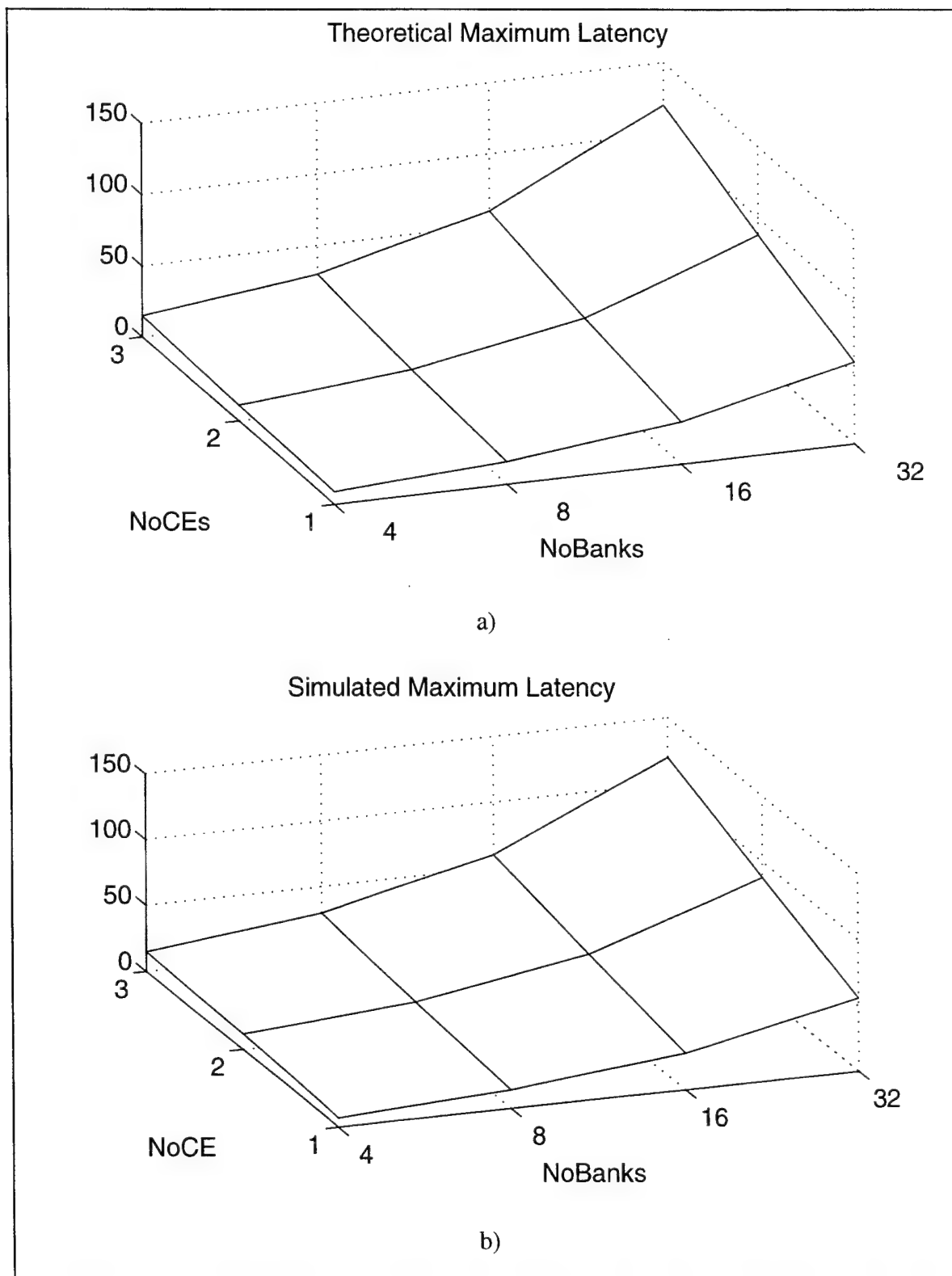
A comparison of the theoretical and simulated steady-state throughput and maximum latency for stride of four is shown in Figure VI.36 and Figure VI.37 respectively. The results are similar to that for stride=2. The simulated steady-state throughput varies by less than two percent from the theoretical results and the simulated and theoretical maximum latencies are identical.

A comparison of the theoretical and simulated steady-state throughput and maximum latency for stride=8 is shown in Figure VI.38 and Figure VI.39, respectively. The results are also similar to those above. However, the simulated steady-state throughput as well as the maximum latency is identical to the theoretical results.

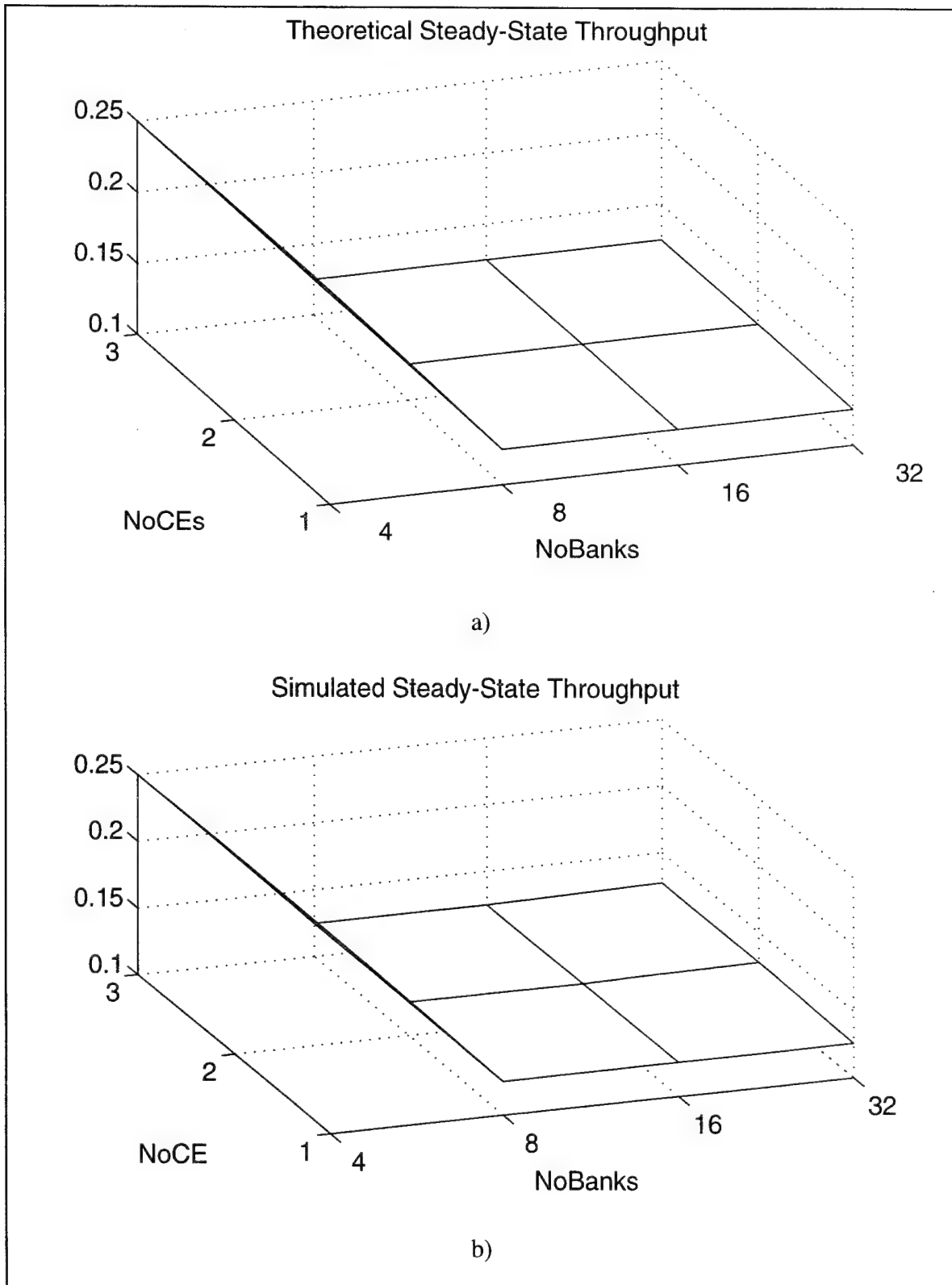


**Figure VI.36 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=4 (Conventional Decoding)**

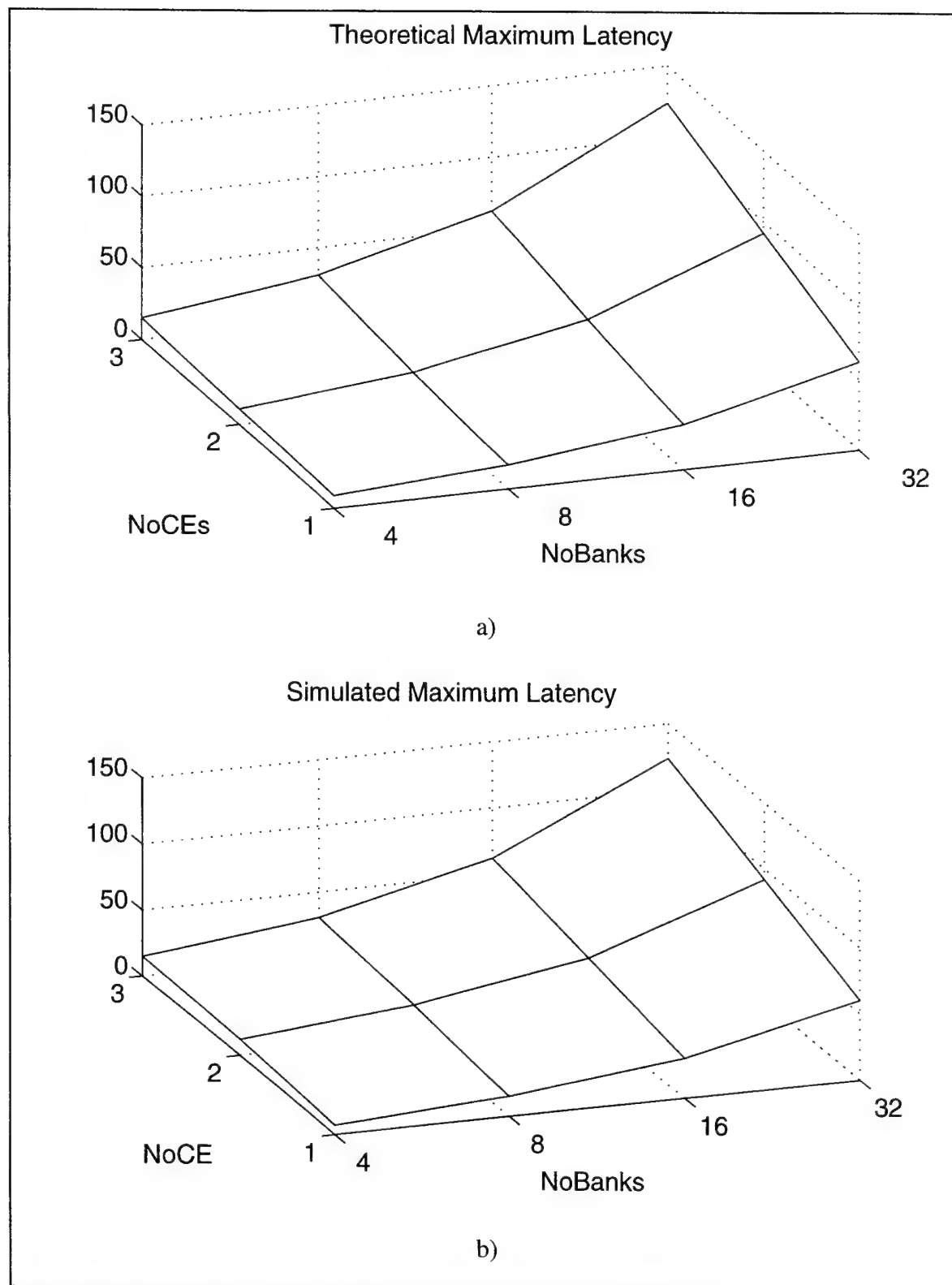




**Figure VI.37 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=4 (Conventional Decoding)**



**Figure VI.38 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=8 (Conventional Decoding)**



**Figure VI.39 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=8 (Conventional Decoding)**

In conclusion, the first experiment verifies the theoretical expressions for steady-state throughput and maximum latency for constant-stride address patterns. These address patterns illustrate optimal results from interleaved memory systems as well as substantial performance degradation when the stride is not relatively prime to the number of banks. STM memories and standard interleaved memories generally have equivalent steady-state throughput performance for constant-stride address patterns when compared to standard interleaving. Further, when the stride is not relatively prime, STM memories incur more latency than standard interleaving.

For the architecture presented in Chapter 0 for FFT computation, those strides that are not relatively prime are the strides required rather than those that are relatively prime. The following experiment is used to validate performance when strides are not relatively prime when permutation-based decoding is used.

## **2. Constant Stride: Permutation-Based Memory Decoding**

An analysis of the second experiment will be divided into strides that are a power of two (e.g., one, two, four,...) and a selected set of strides not a power of two (e.g., three and five). It is important to recall that the estimates for the maximum latency for permutation-based memories are always upper-bound estimates. The theoretical steady-state throughput results for the radix- $r$  butterfly and digit-reversed address patterns are also lower bounds. However, for address patterns with constant stride, the theoretical results are exact for the STM cases (i.e., when the number of cache elements is greater than one).

A comparison of the theoretical and simulated results for strides of one, two, and sixty four are shown in Figure VI.40 through Figure VI.45. This selection of strides is presented as a representative of all of the strides of powers of two possible, given the permutation matrices used in the simulation and documented in Figure III.13 through Figure III.16.

The most striking characteristic of the plots contained in Figure VI.40 through Figure VI.45 is that they are for all practical purposes the same. Plots for stride of one are shown in Figure VI.40 and Figure VI.41. The simulated steady-state throughput is

identical to the theoretical for all STM cases (they are always optimum). The theoretical standard interleaving cases are characterized by a decreasing lower-bound steady-state throughput as the number of banks increases. Recall that this lower bound is based on the possibility that a bank will receive two consecutive requests at the boundaries of the base sequences (refer to Chapter V Section D for a discussion of this topic). Figure VI.40 suggests that with the permutation matrices used, this is not the case. Further, as the number of banks increases, the probability decreases.

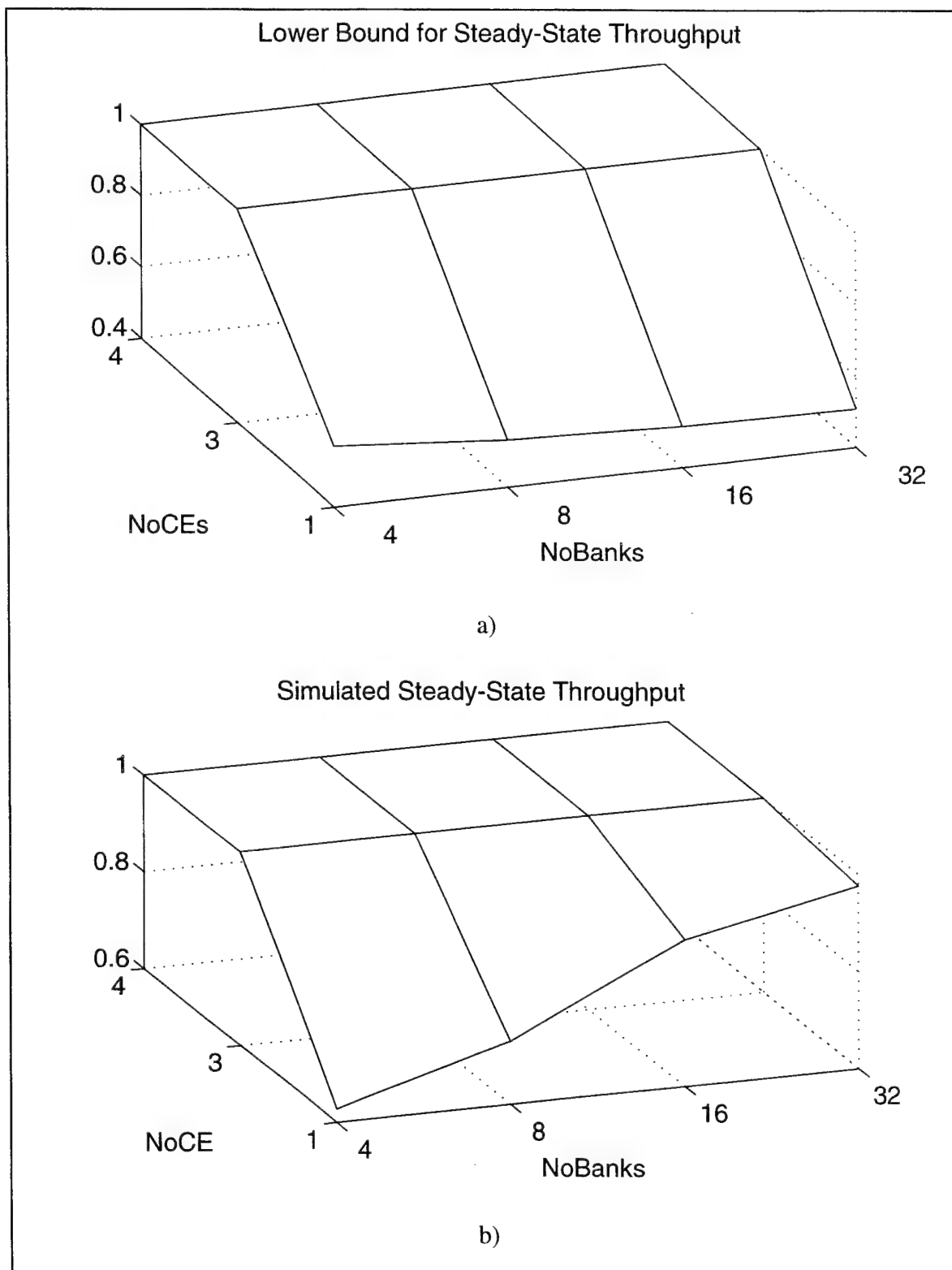
Several observations can be made concerning the theoretical and simulated maximum latencies shown in Figure VI.41. First, the basic shape of the theoretical and simulated maximum latencies are similar in that they both increase with the number of banks and are relatively invariant to the number of cache elements. The simulated maximum latency is, however, equal to the theoretical maximum for four-bank memory with a substantial differential between them for the 32-bank memory. This is due to the relative length of the input vector to the number of banks. This issue will be explored more fully in the permutation-based digit-reversed experiment below.

Examples of two simulation runs, the first with four banks and three cache elements and the second with thirty two banks and three cache elements are shown in Figure VI.46 and Figure VI.47, respectively. It is not possible to anticipate when latency will be incurred. For a small number of banks, it is more likely that the maximum latency will be incurred earlier than in a memory configured with more memory banks because the likelihood of two banks being close is greater when the number of banks is small. Observe the distribution of the latency in Figure VI.46 versus Figure VI.47. In the first plot with four banks, the maximum latency is incurred early in the run in contrast to the thirty two bank simulation where the maximum latency (in the plot) is not obtained until half way through the simulation run. In Figure VI.48, the simulation is run with an input vector of 1,024 and it can be seen that the maximum latency is not reached until approximately cycle 600.

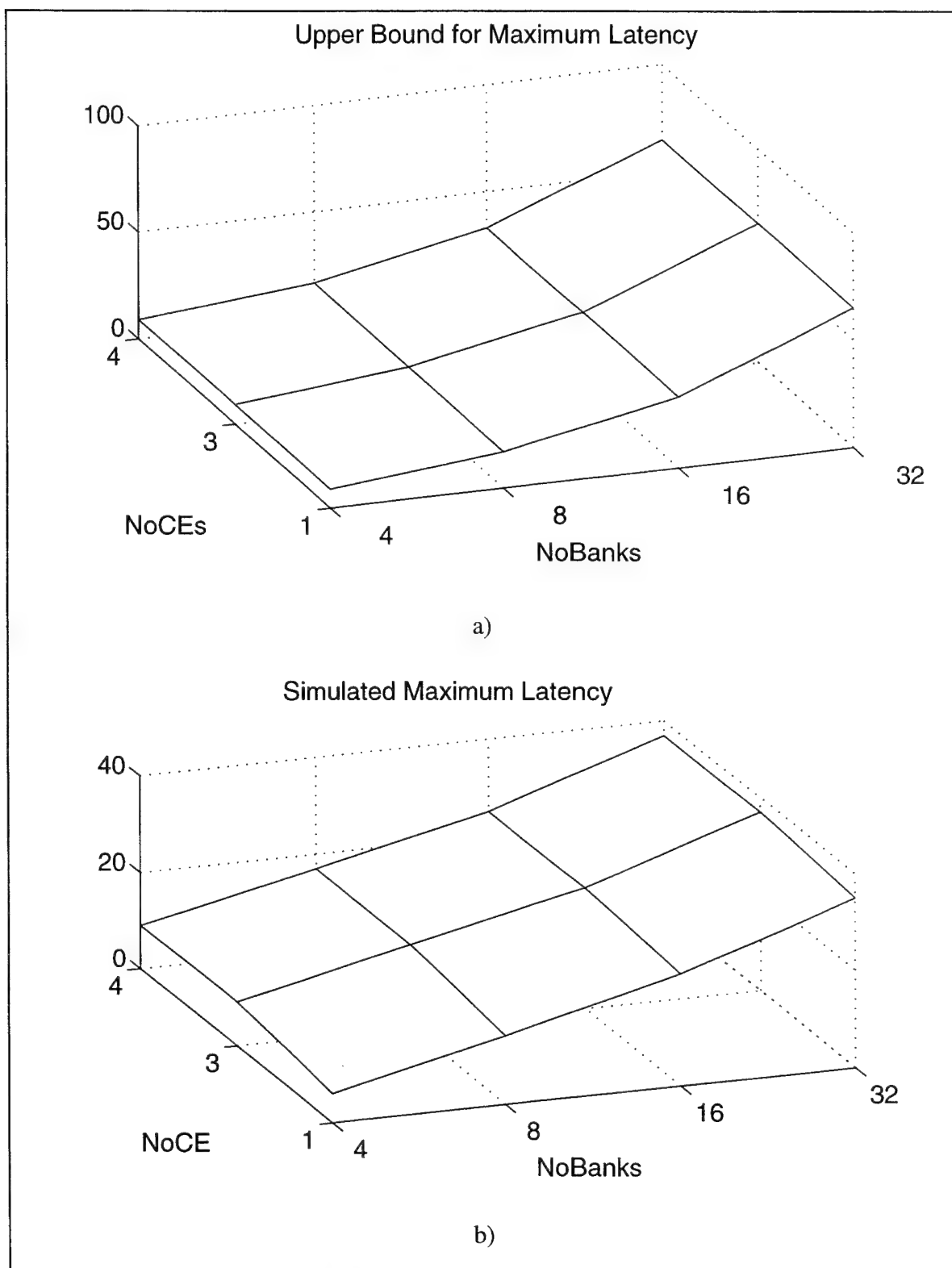
The tradeoff between latency and the number of memory banks can be seen in the detailed plots by the amount of time required to obtain the first memory response. The

ratio of time between the minimum latency and the length of the input vector dictates the best average throughput (i.e., the greater the ratio, the greater the penalty). This latency results in a average throughput of 0.78, as shown above the throughput plot in Figure VI.47. This reflects a modest increase in latency from the conventional decoding case. Figure VI.49 illustrates the effect of varying the number of banks on average throughput for the case of stride=64. The penalty of a larger number of banks is clearly shown for STM memories when the vectors are relatively small. In general, STM performance is better than standard interleaving except when the number of banks is 32, where the performance is approximately the same.

The steady-state throughput and average throughput for strides of three and five are illustrated in Figure VI.50 and Figure VI.51. Clearly the throughput is diminished from the powers of two cases shown above. The steady-state throughput for stride of three falls steadily as the number of banks increases, whereas the stride of five case has the same steady-state throughput for four and 32 banks but reduced throughput for 16 banks. These figures seem to confirm the erratic behavior of permutation-based performance for strides that are relatively prime to strides of two. The average throughput for four and eight bank systems suggest moderate performance that might be tolerated if the address pattern was not frequently used.

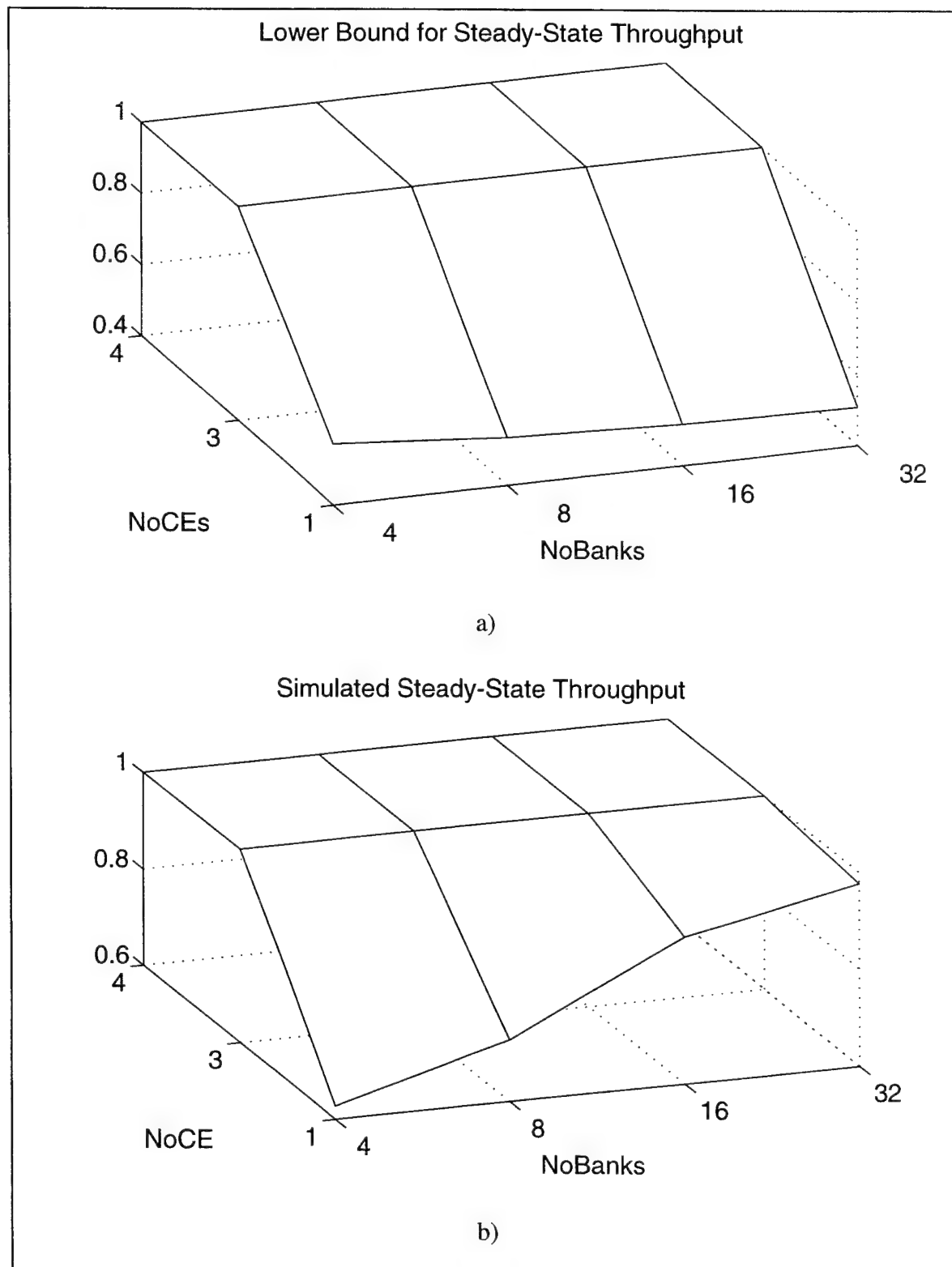


**Figure VI.40 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=1 (Permutation-Based Decoding)**

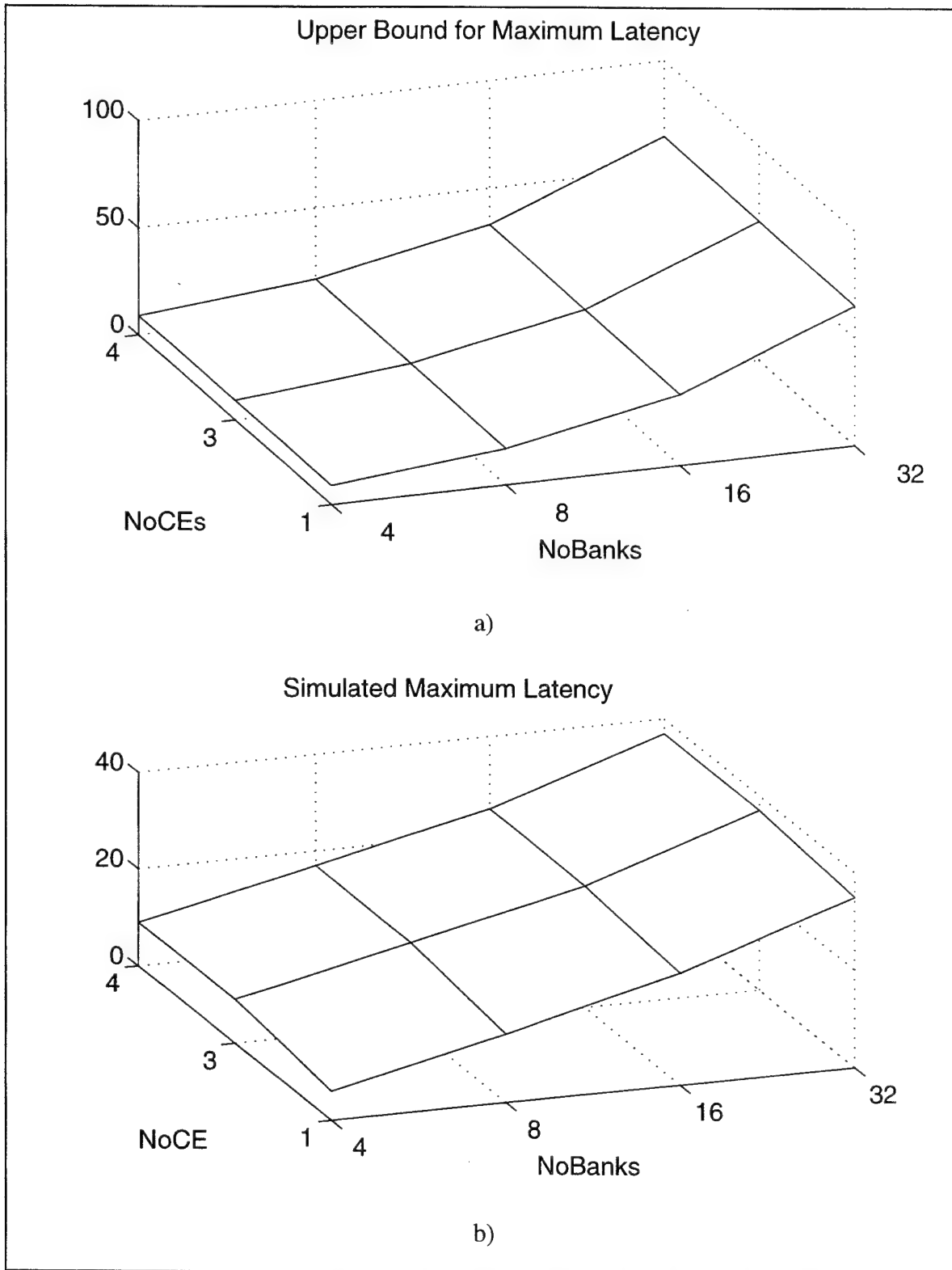


**Figure VI.41 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=1 (Permutation-Based Decoding)**

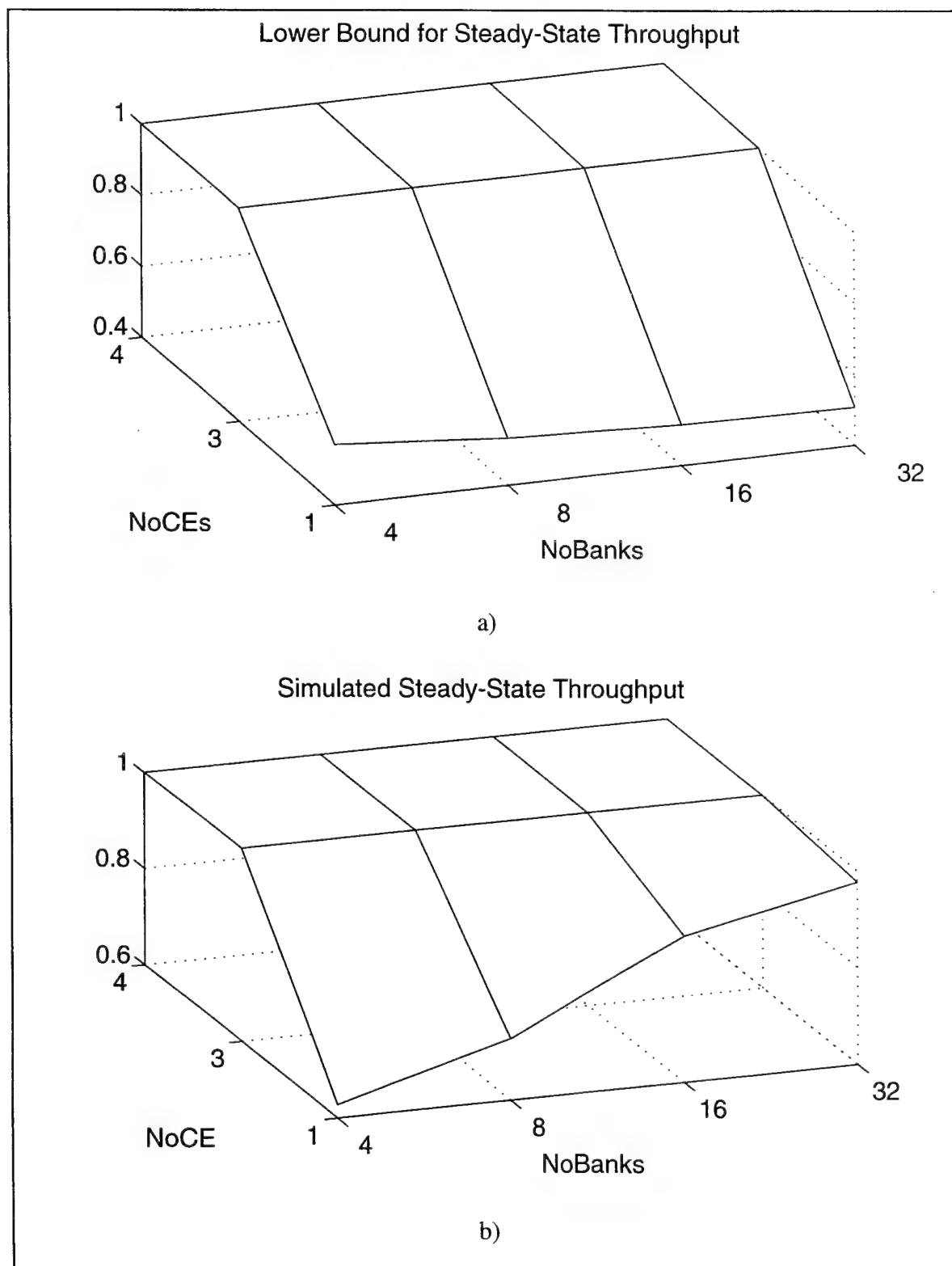




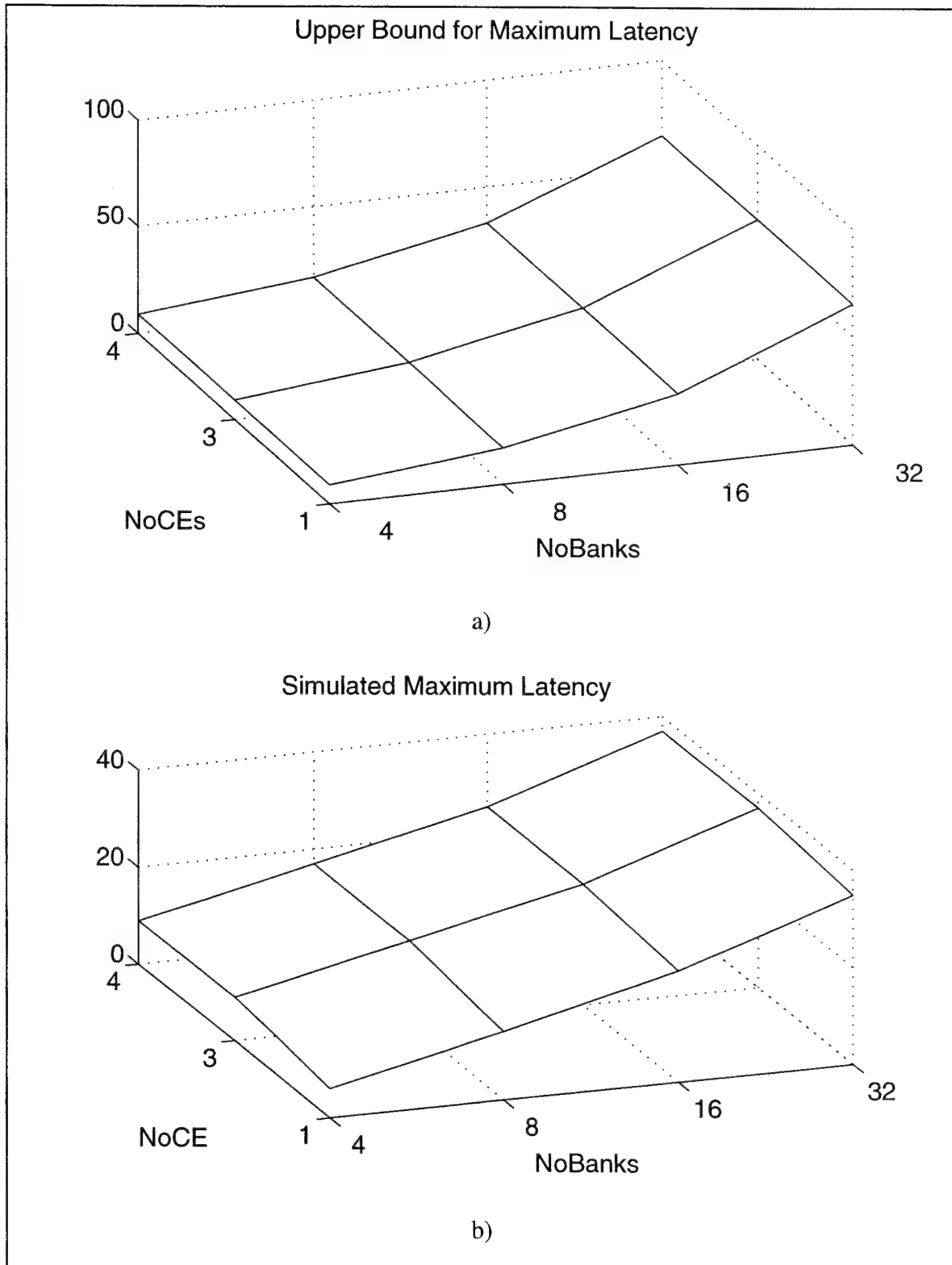
**Figure VI.42 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=2 (Permutation-Based Decoding)**



**Figure VI.43 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=2 (Permutation-Based Decoding)**



**Figure VI.44 Comparison of Theoretical Versus Simulated Steady-State Throughput for Stride=64 (Permutation-Based Decoding)**



**Figure VI.45 Comparison of Theoretical Versus Simulated Maximum Latency for Stride=64 (Permutation-Based Decoding)**

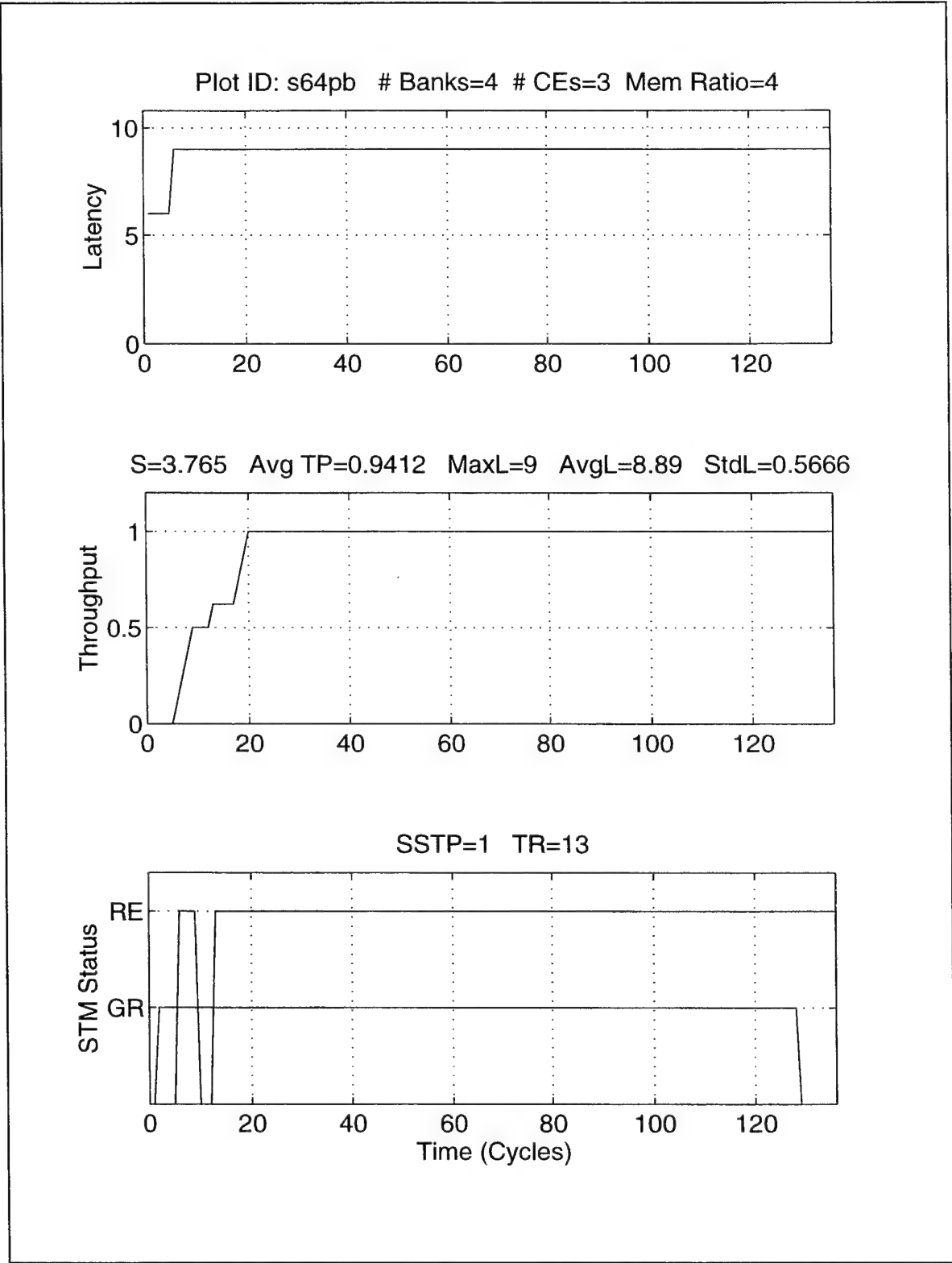


Figure VI.46 Detailed Simulation Run for Stride=64 STM(4,3,4) (Permutation-Based Decoding)

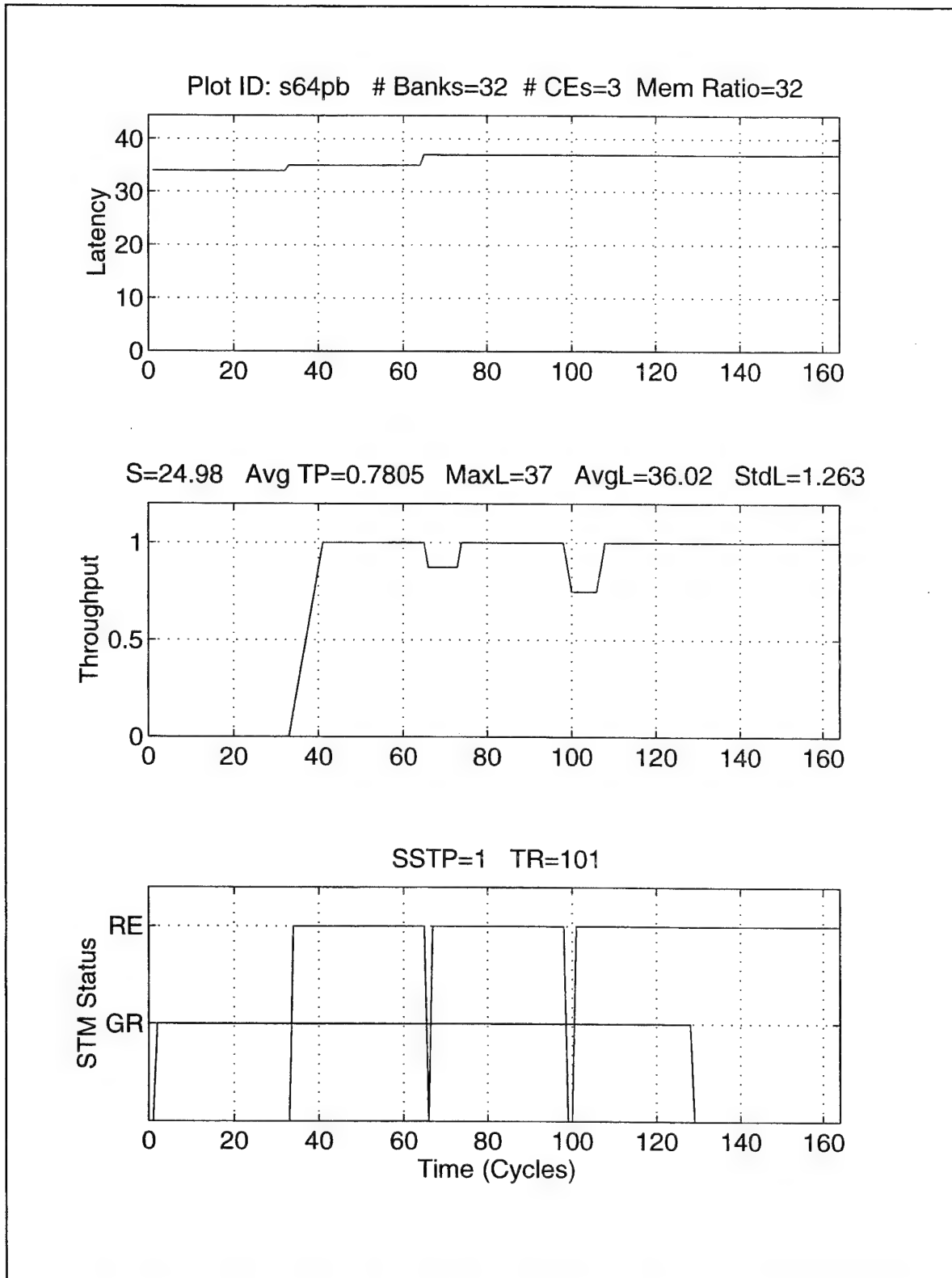
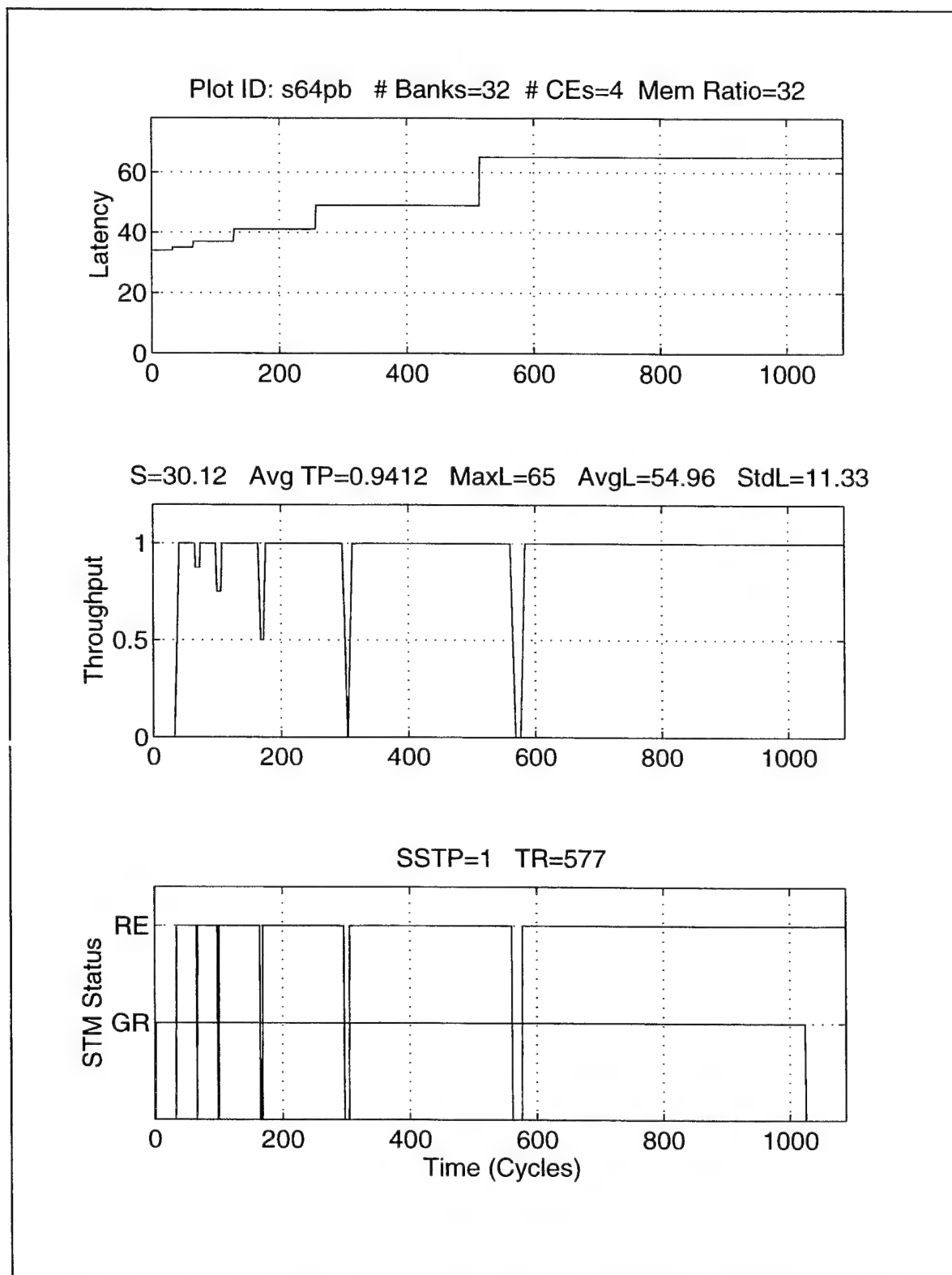
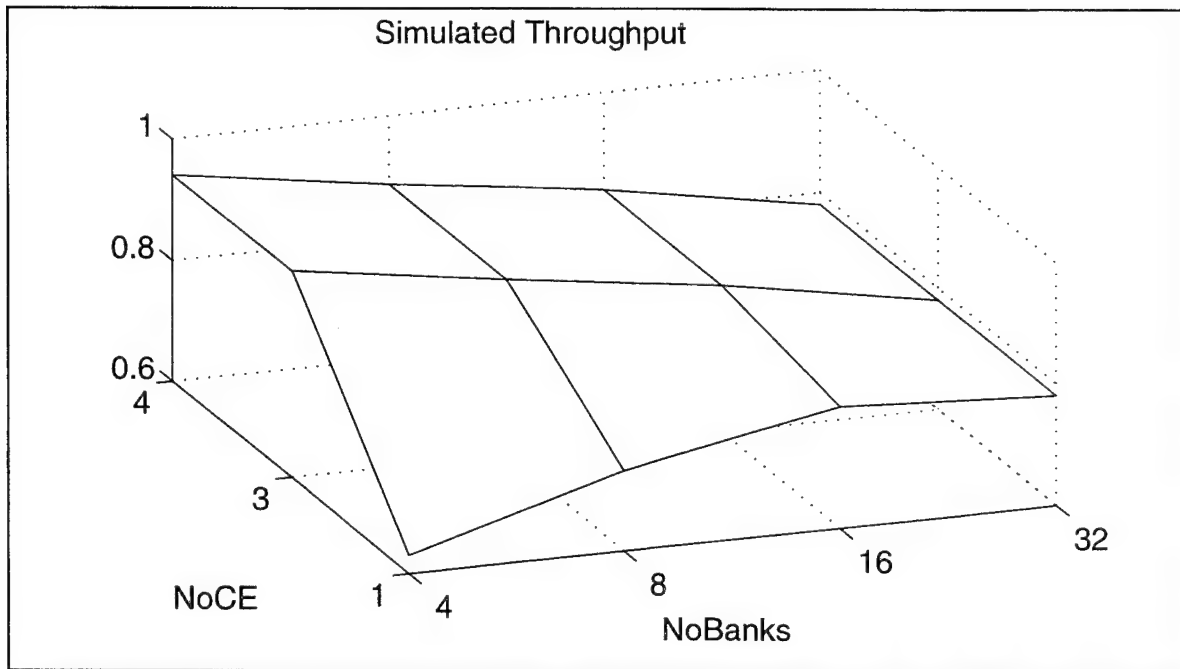


Figure VI.47 Detail Simulation Run for Stride=64 STM(32,3,32) (Permutation-Based Decoding)



**Figure VI.48 Second Detailed Simulation Run for Stride=64 STM(32,3,32)  
(Permutation-Based Decoding)**

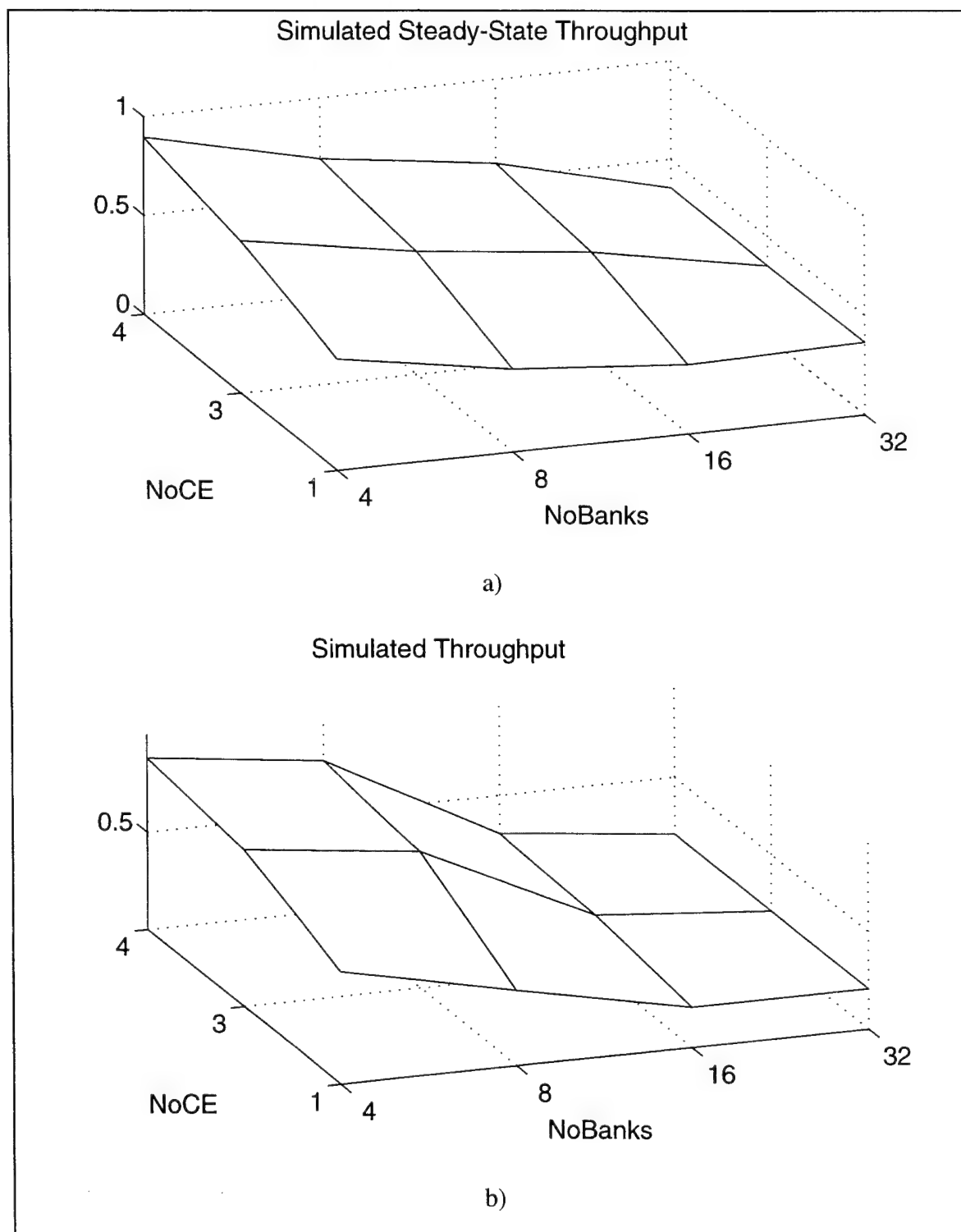


**Figure VI.49 Simulated Average Throughput for Stride=64 (Permutation-Based Decoding)**

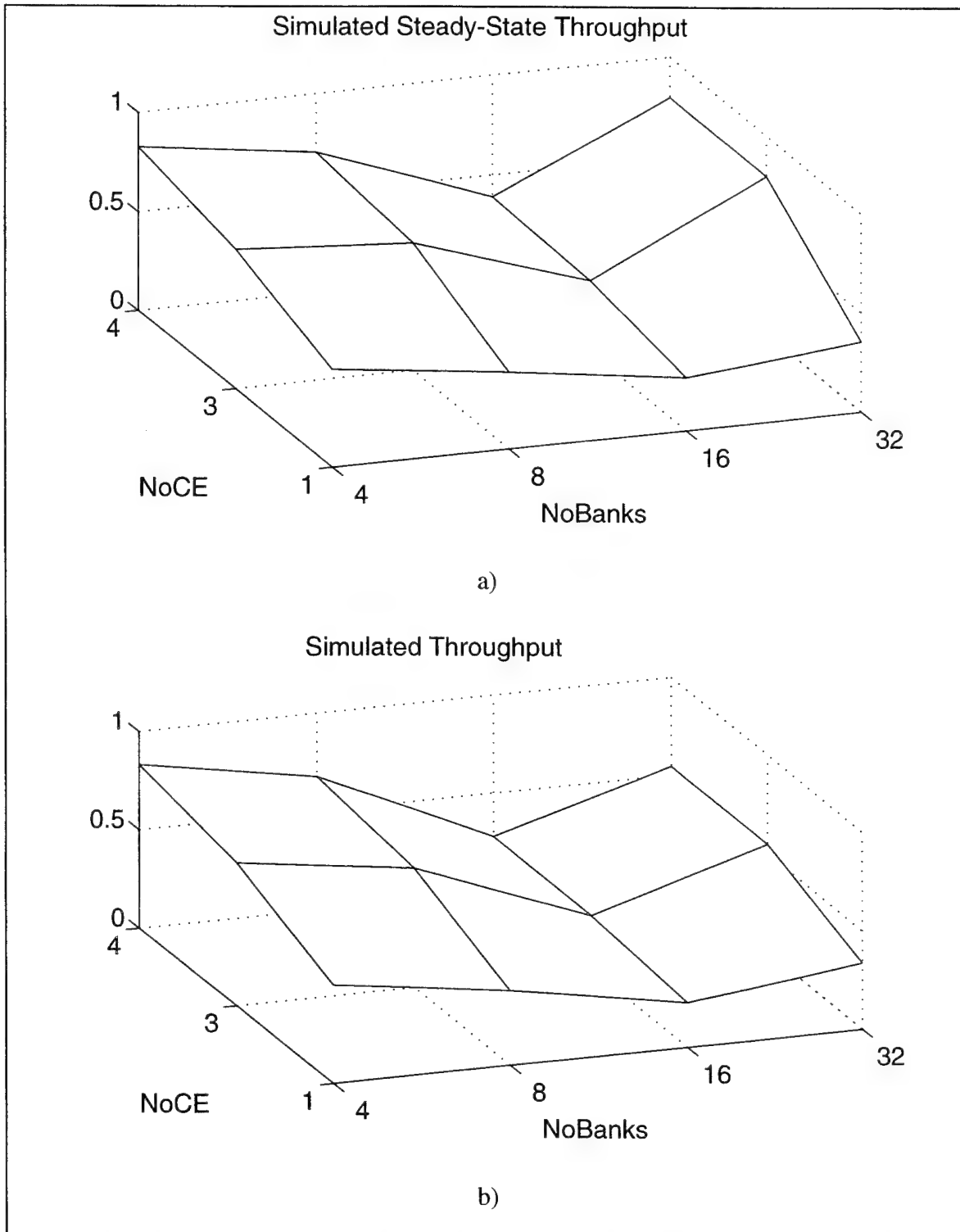
This experiment demonstrates that permutation-based decoding can provide favorable performance for address patterns with a constant stride of powers of two. Specifically, the steady-state throughput is optimum when the number of cache elements is at least three. Further, this is accomplished with a modest increase in the latency when the vector length is small (e.g., 128 in the examples). For larger vector lengths where the maximum latency is realized, the increase in the latency is approximately doubled.

The next section will address radix- $r$  butterfly address patterns when conventional decoding is in place.





**Figure VI.50 Simulated Steady-State Throughput and Average Throughput for Stride=3 (Permutation-Based Decoding)**



**Figure VI.51 Simulated Steady-State Throughput and Average Throughput for Stride=5 (Permutation-Based Decoding)**

### 3. Radix-r Butterfly: Conventional Memory Decoding

A comparison of the theoretical versus simulated steady-state throughput and maximum latency are shown in Figure VI.52 through Figure VI.59 for radices two, four, eight, and 16. The simulated steady-state throughput is in agreement with the theoretical results for all radices. An inspection of Figure VI.52 reveals that all STM memories yield a throughput of 1.0 as expected. Standard interleaving cases suffer significant degradation because each bank is given consecutive memory requests equal to the radix. The greater the number of banks, the more banks there are not performing as indicated for all radices.

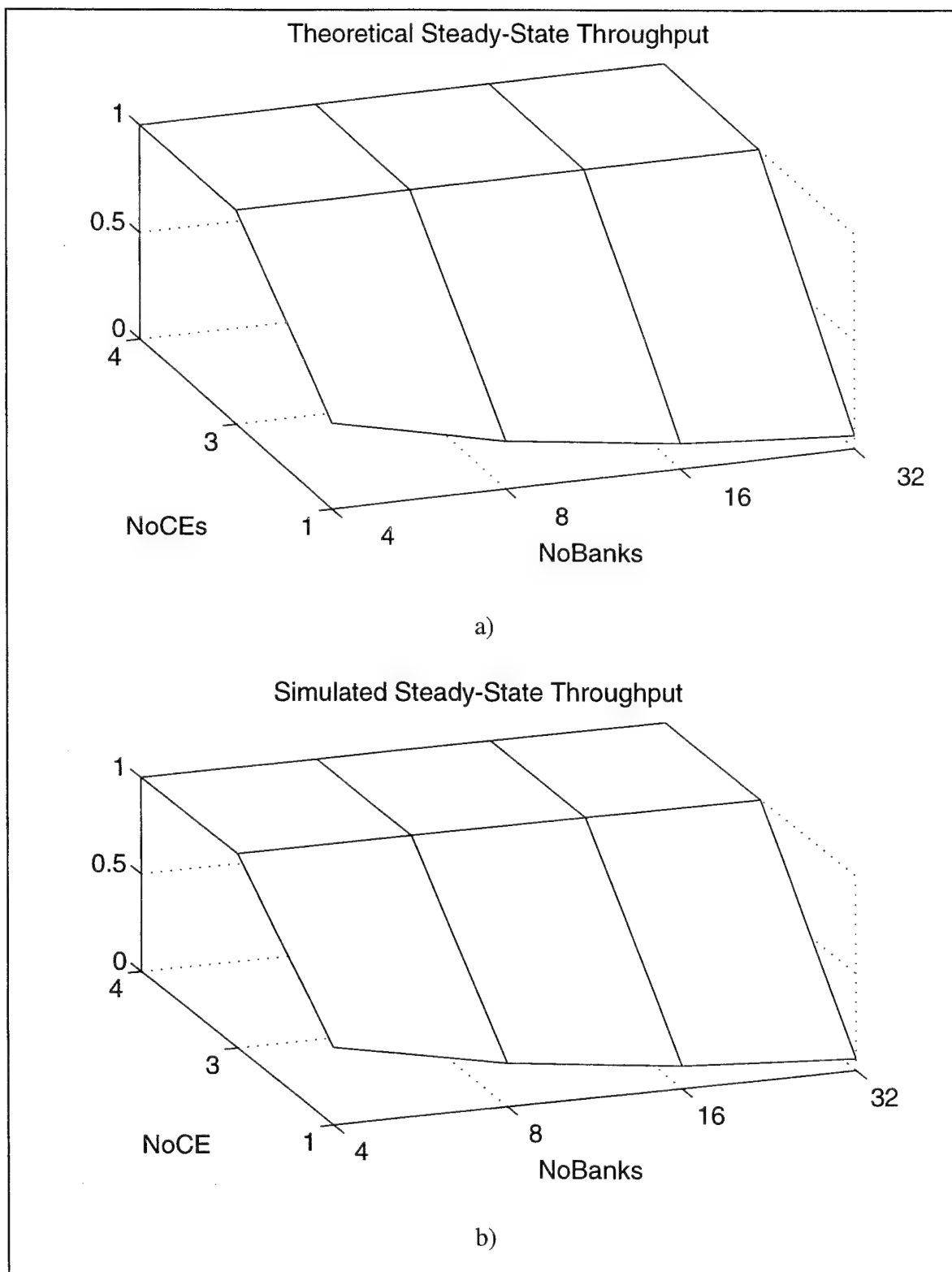
The simulated maximum latency is in complete agreement for the radix-2 and radix-4 cases as shown in Figure VI.53 and Figure VI.55, respectively. However, variances occur for both the radix-8 (32 banks) and radix-16 (16 and 32 banks) cases. In both cases, the maximum latency rather than continuing to rise as the expressions would suggest, flatten out. This phenomena is due to the relationship between the “stride” in effect for radix- $r$  patterns and the expression for the effective number of banks. Whenever the effective stride becomes smaller than the number of banks, then the latency will be reduced. For example, for the radix-16 case, the effective stride is

$$S = \frac{N}{r} = \frac{2^7}{2^4} = 2^3$$

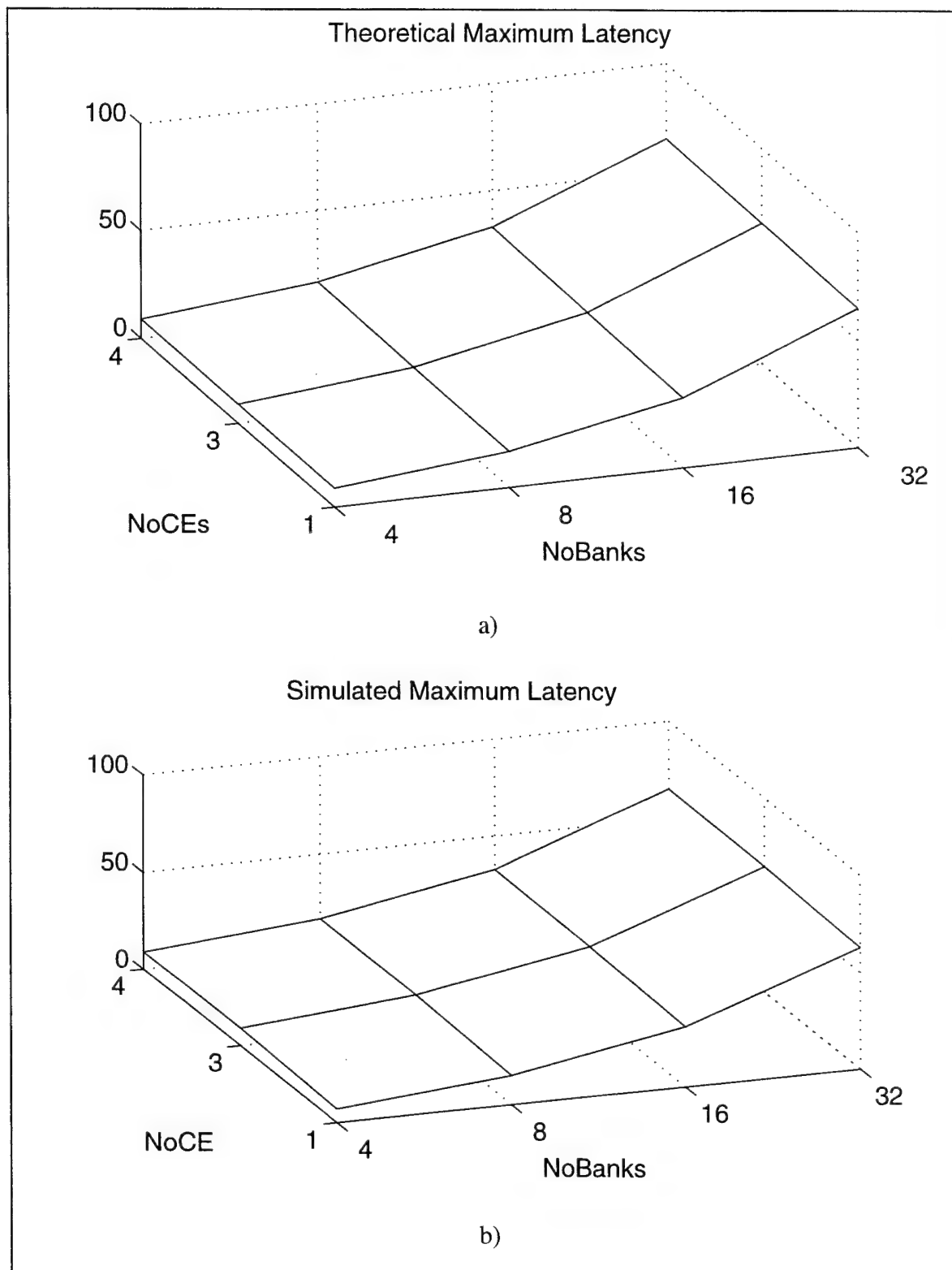
and the effective number of banks is

$$B_{eff} = \frac{B}{\gcd(B, S)} = \frac{16}{\gcd(16, 8)} = 2.$$

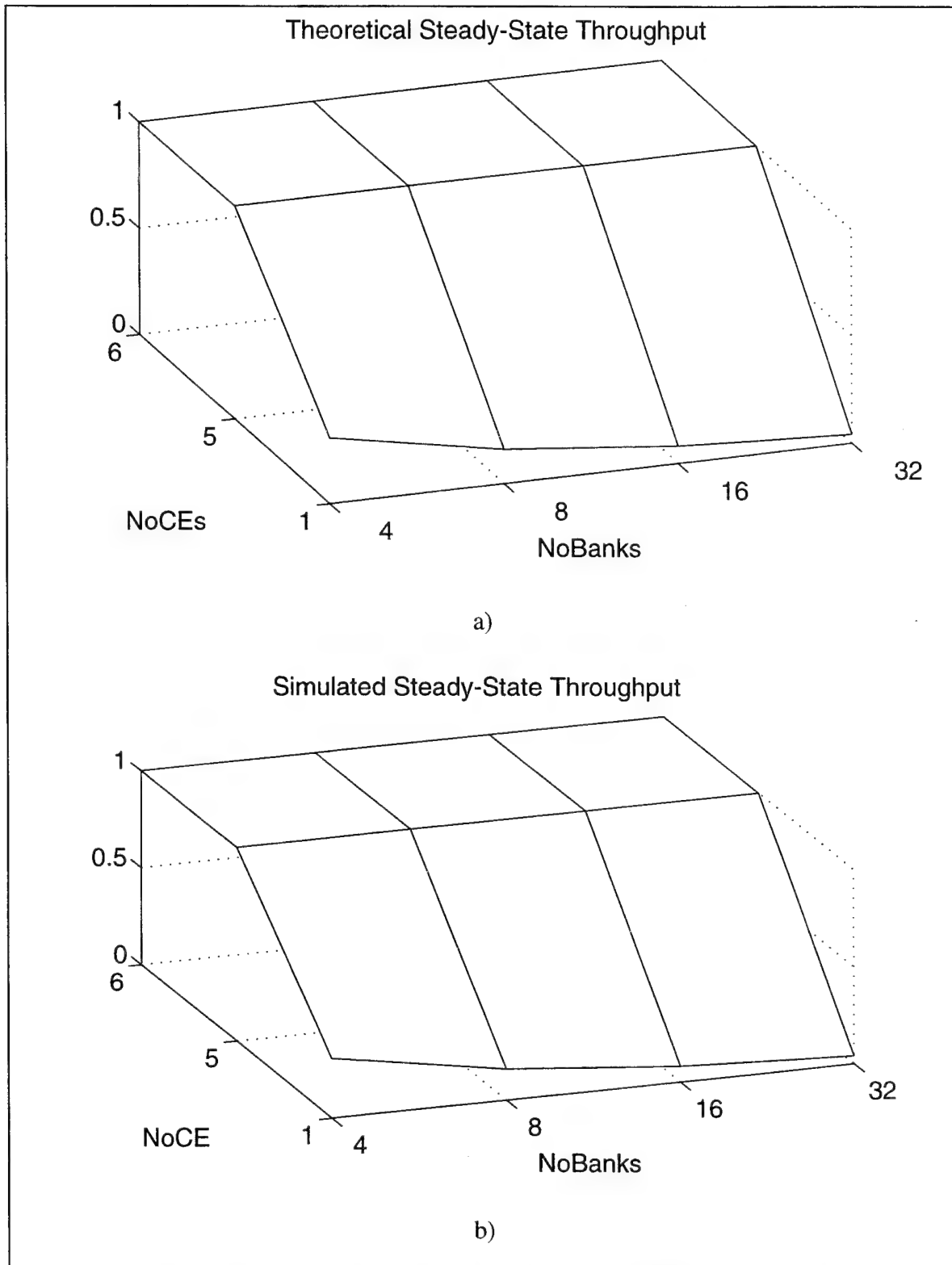
Recall that  $B_{eff}$  under most circumstances is one when the radix and number of banks is a power of two. If  $N$  is doubled, the expression for latency used to construct the theoretical plots will be valid.



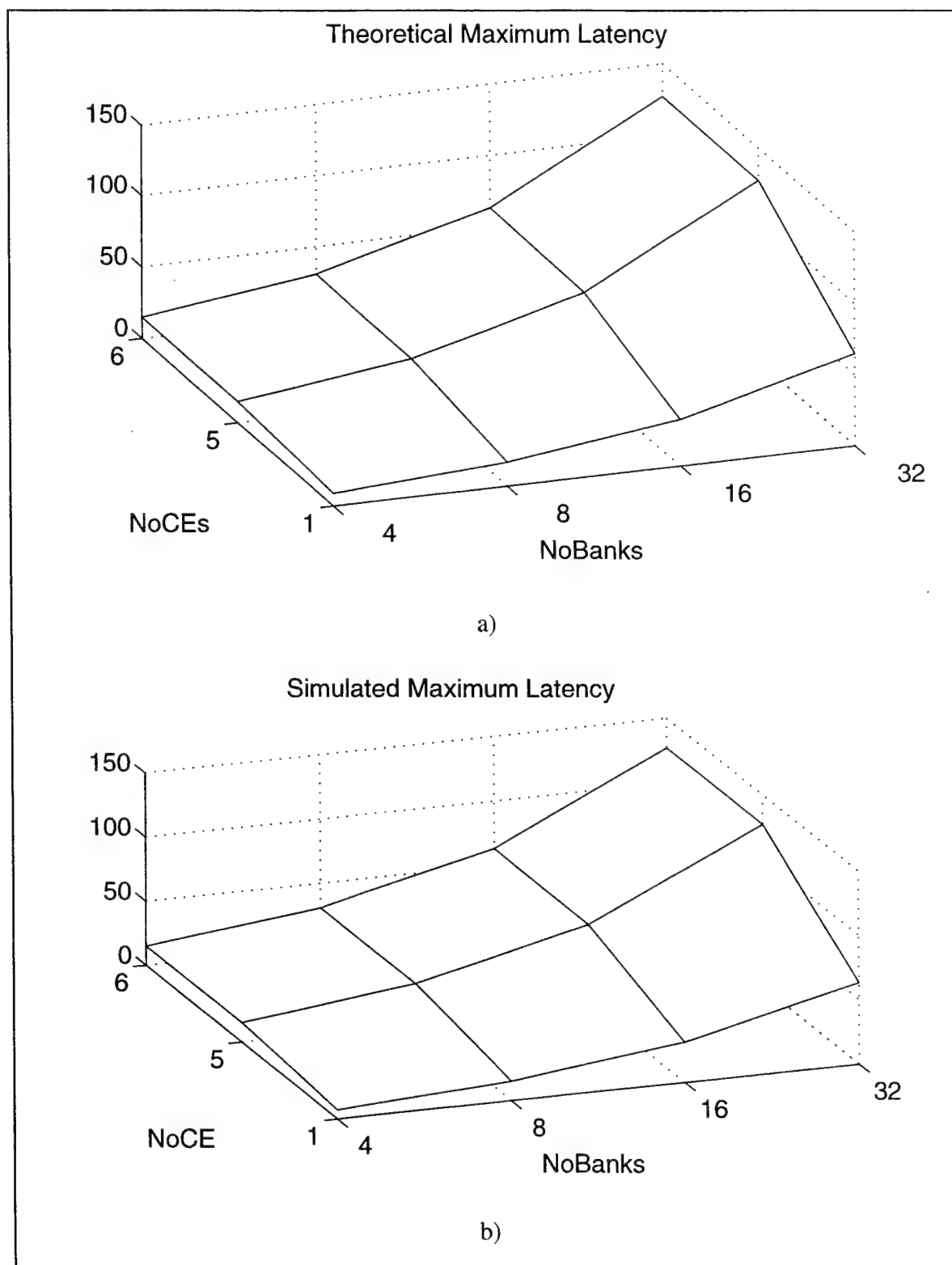
**Figure VI.52 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=2 (Conventional Decoding)**



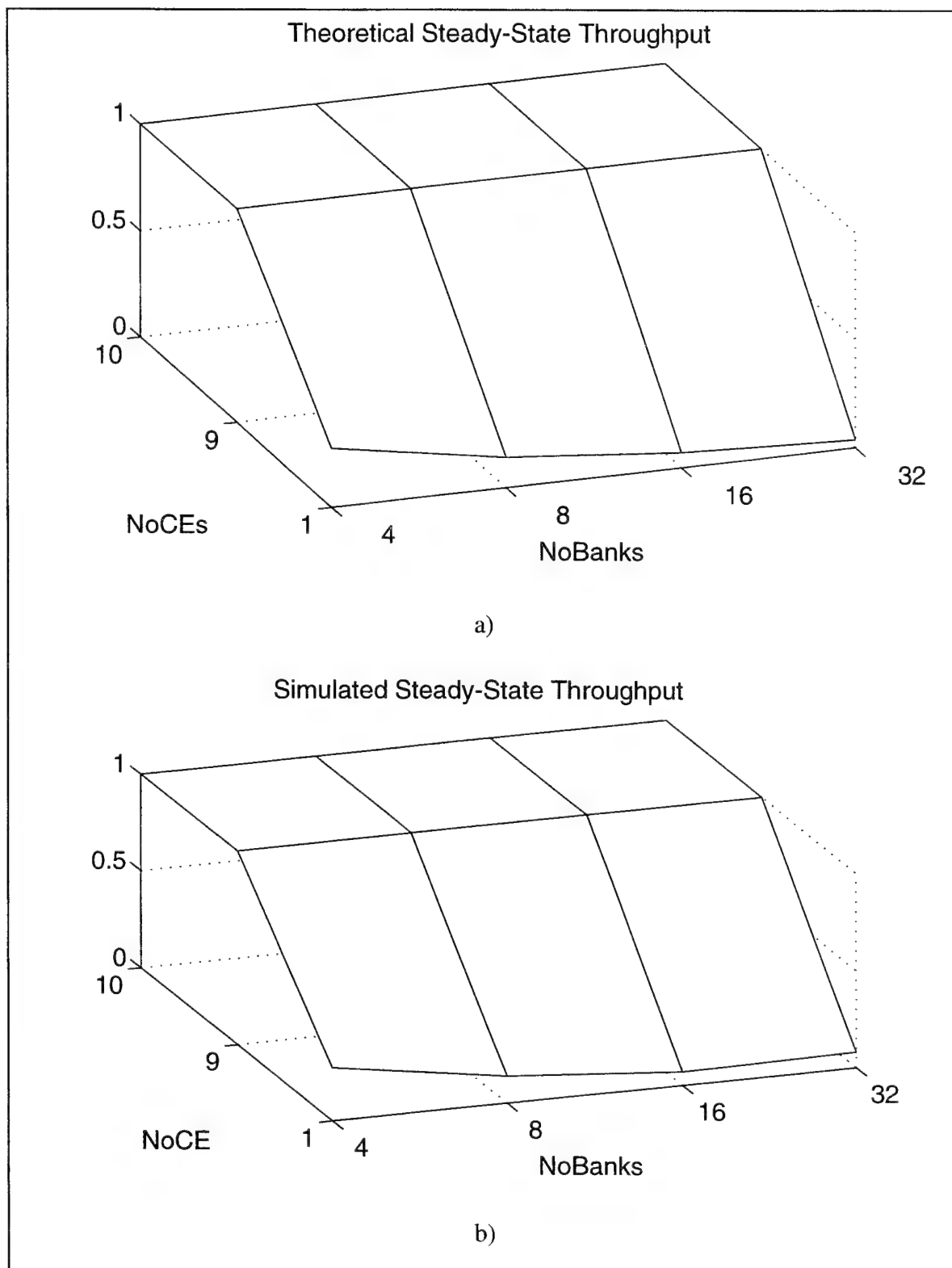
**Figure VI.53 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=2 (Conventional Decoding)**



**Figure VI.54 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=4 (Conventional Decoding)**

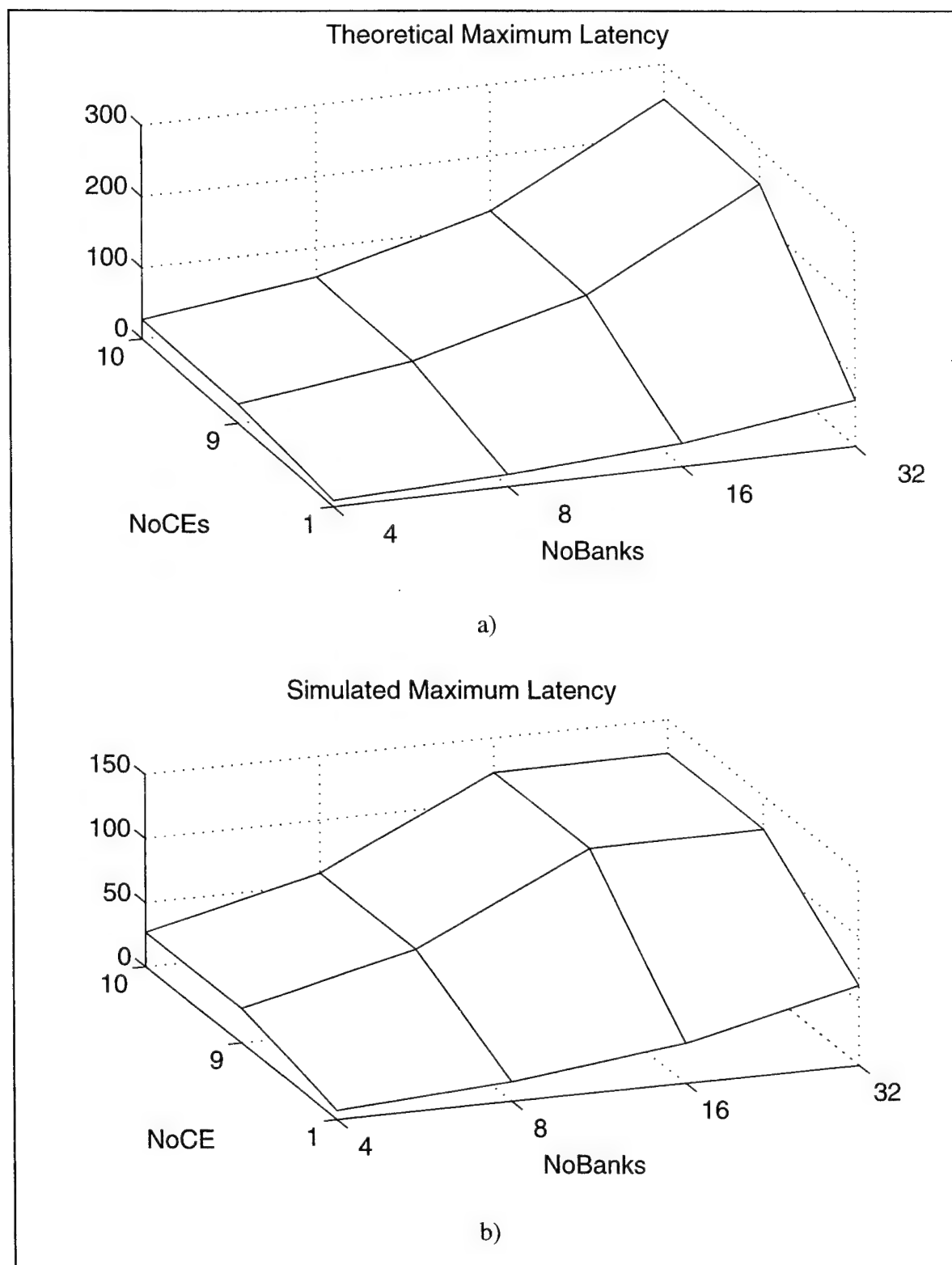


**Figure VI.55 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=4 (Conventional Decoding)**

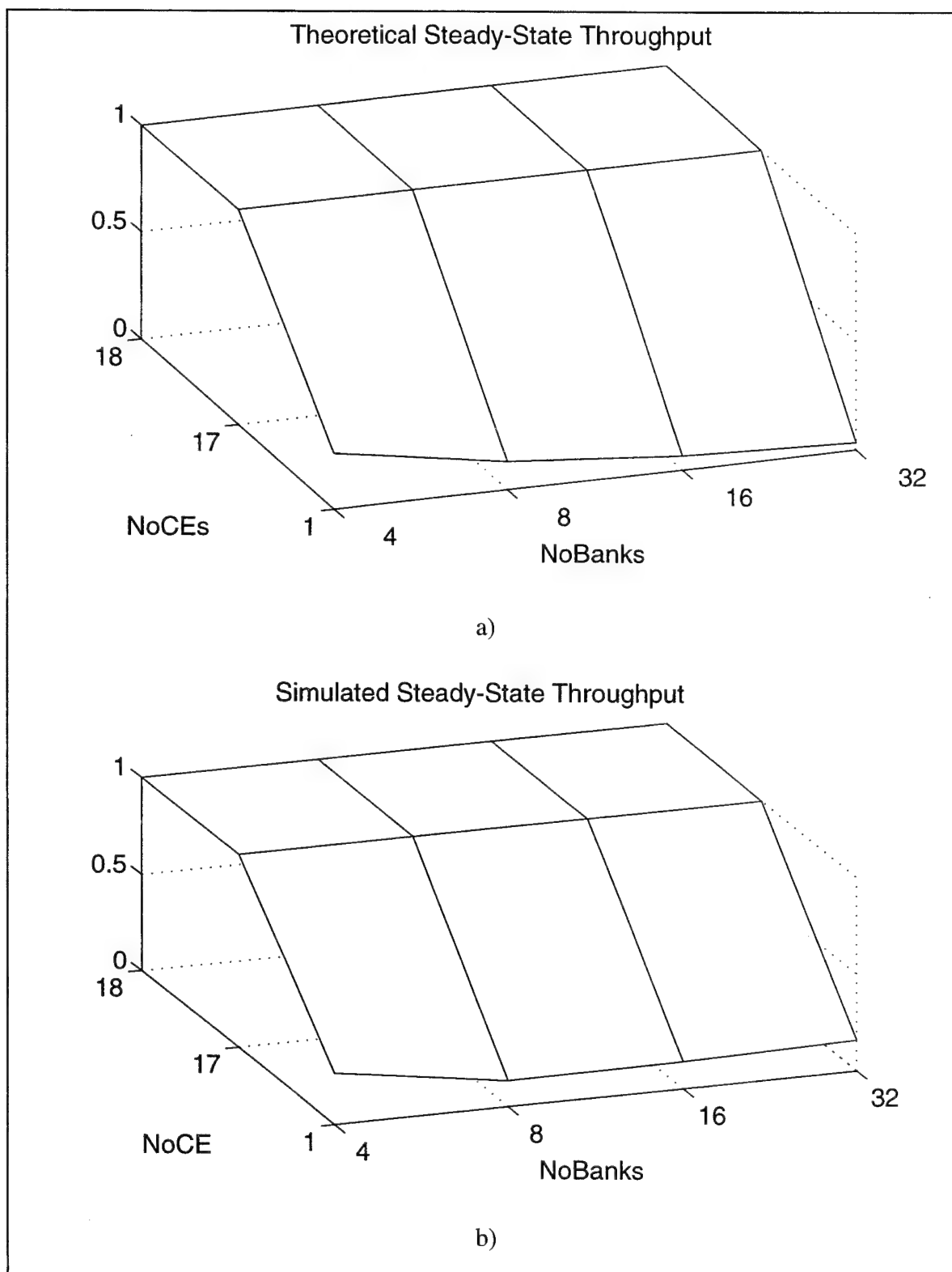


**Figure VI.56 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=8 (Conventional Decoding)**

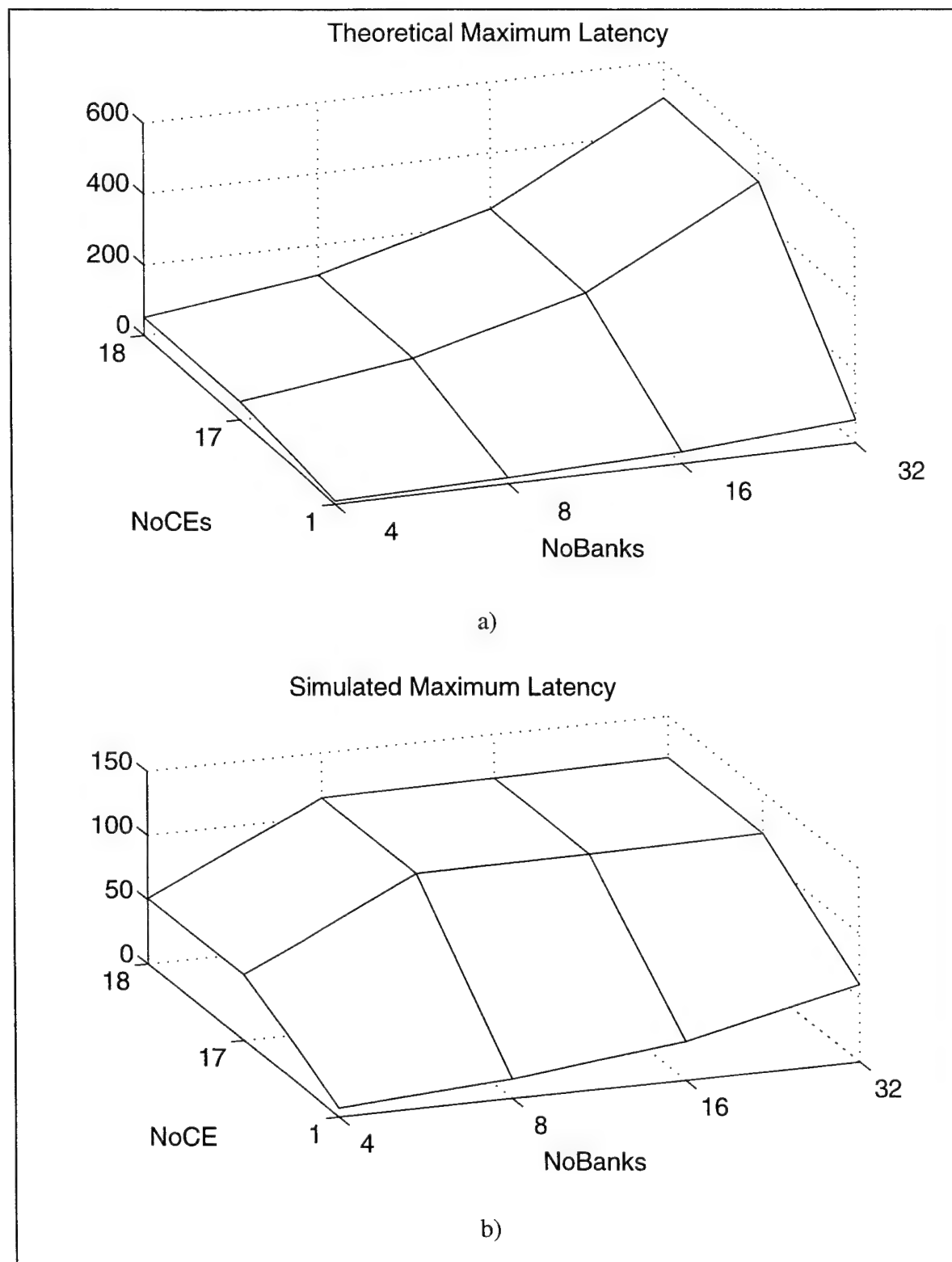




**Figure VI.57 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=8 (Conventional Decoding)**



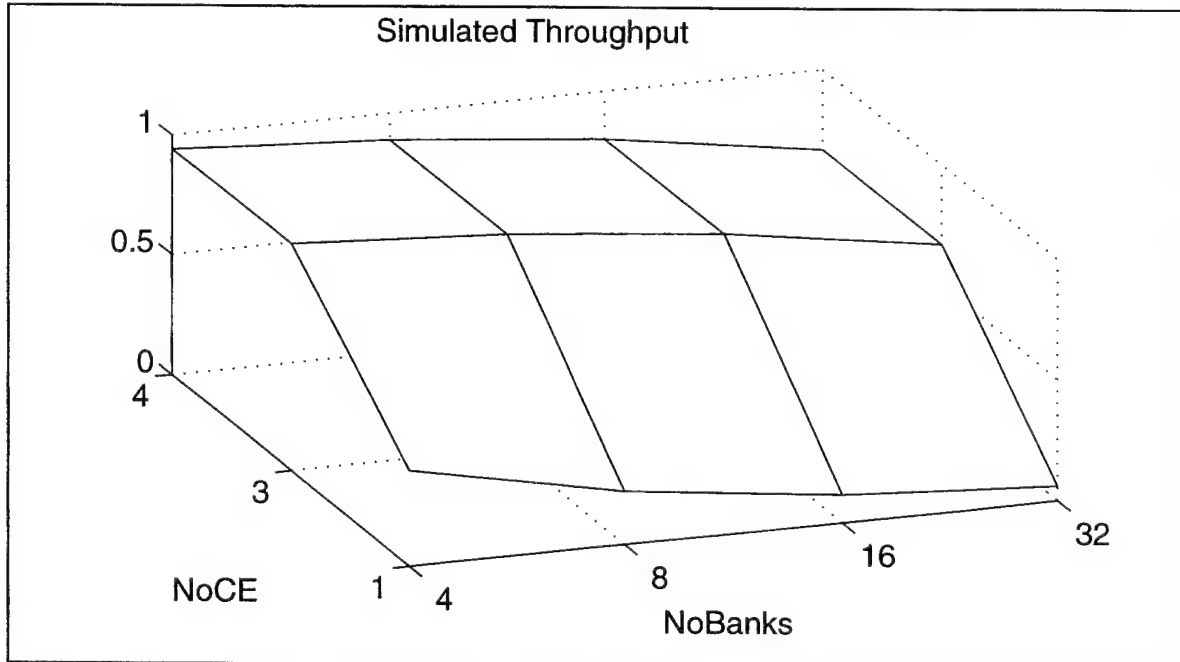
**Figure VI.58 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=16 (Conventional Decoding)**



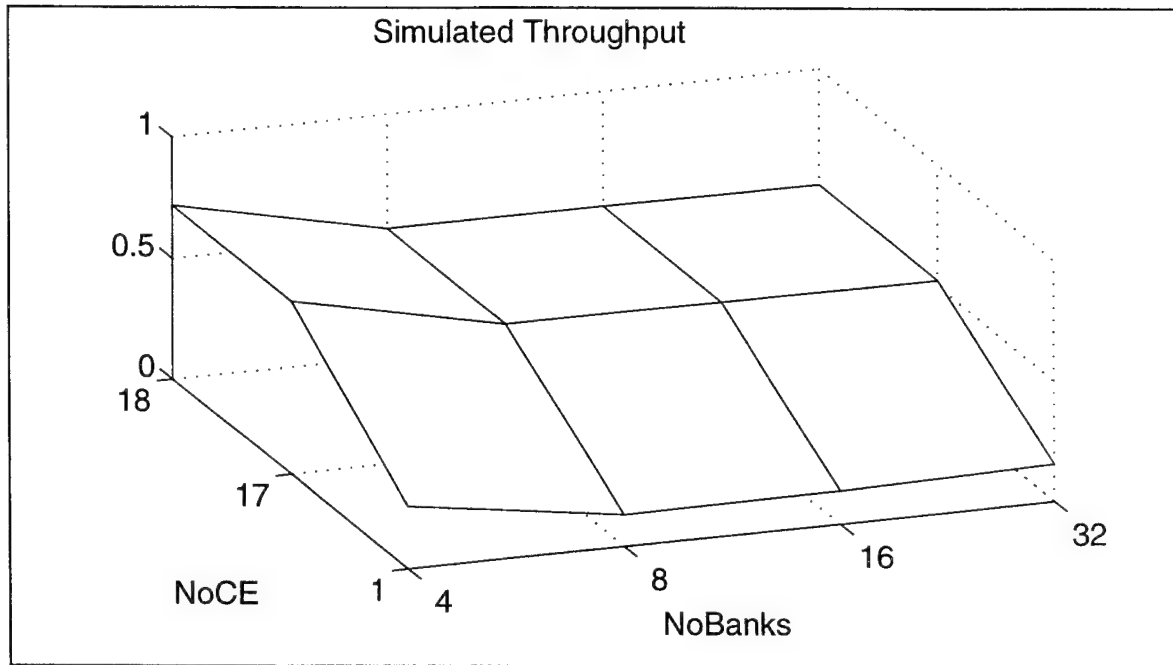
**Figure VI.59 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=16 (Conventional Decoding)**

One last issue to address for conventional decoding of radix- $r$  butterfly address patterns is the average throughput. Average throughput of a radix-2 and radix-16 address patterns are displayed in Figure VI.60 and Figure VI.61. The radix-2 butterfly yields average throughput values between 0.9 and 0.7, approximately five to ten percent lower than the constant stride patterns. The radix-16 butterfly average throughput is considerably worse beginning at 0.7 with a lower end of 0.5. The lower end would, of course, be worse for longer vectors. Therefore, this must be taken into account when calculating the efficiency of the vector processor or when determining the most effective combination of radix passes to use for a given length vector.

This experiment validates that conventional decoding coupled with STM with a sufficient number of banks will provide an optimum throughput, but at higher latencies than encountered with either conventional or permutation-based decoding of constant strides. The following section will investigate permutation-based decoding of radix- $r$  address patterns.



**Figure VI.60 Average Throughput for Radix-2 Butterfly Pattern (Conventional Decoding)**



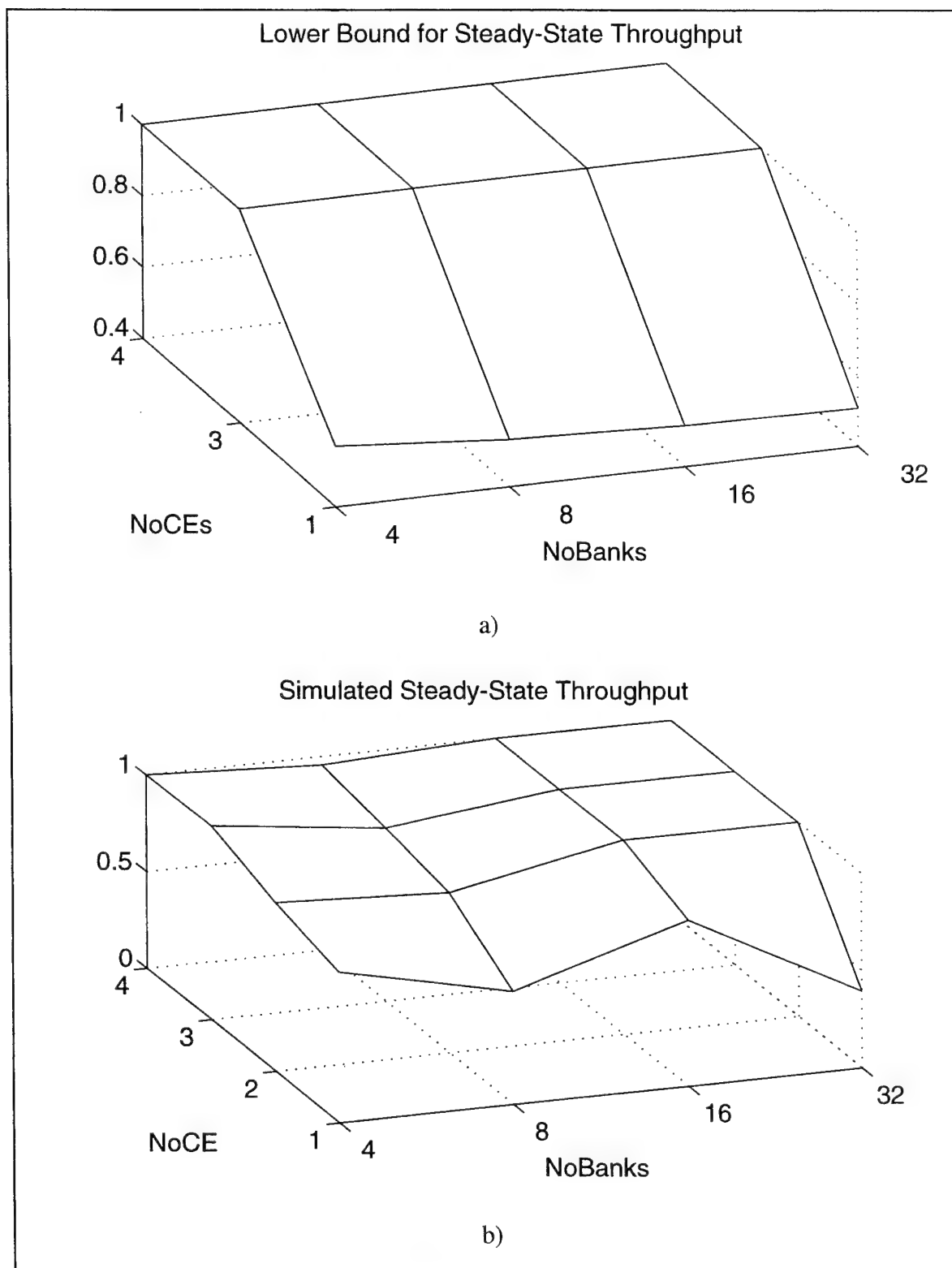
**Figure VI.61 Average Throughput for Radix-16 Butterfly Pattern (Conventional Decoding)**

#### **4. Radix-r Butterfly: Permutation-Based Memory Decoding**

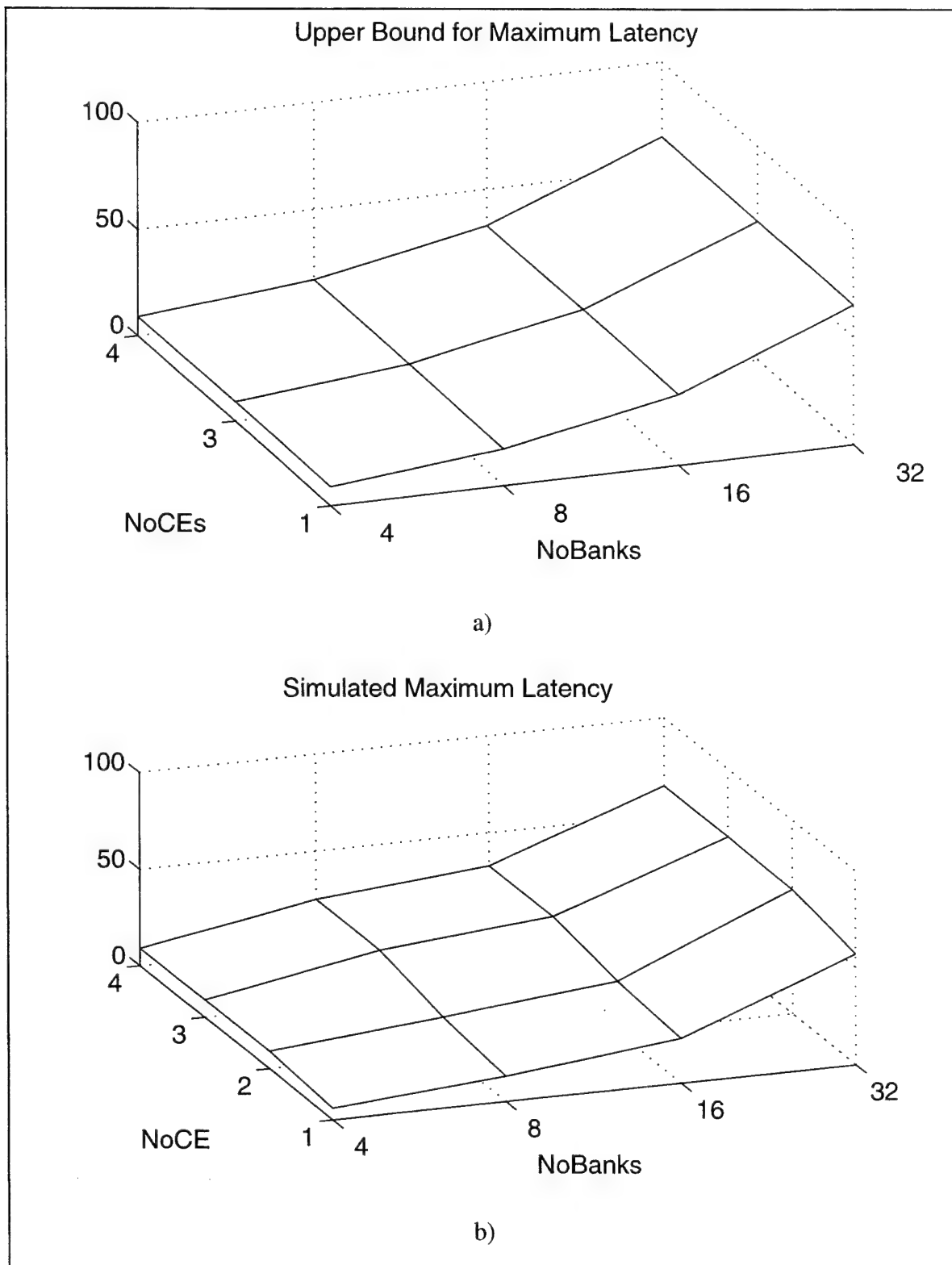
The theoretical and simulated results for the steady-state throughput and maximum latency are illustrated for radices two, four, eight, and 16 in Figure VI.62 through Figure VI.71. The theoretical steady-state results are lower bounds for the standard interleaving case. The theoretical maximum latency is an upper bound for all values. The simulation runs were executed with the tailored permutation matrices for radices four, eight, and 16. Although it is possible to develop a tailored permutation matrix for radix 2, radix 2 patterns yield good performance without it. Further, there are operational constraints that make it desirable not to have a specialized permutation matrix for radix 2. For more details, see the conclusions in Chapter VII.

The simulated values for steady-state throughput and maximum latency are shown in Figure VI.62 and Figure VI.63, respectively. The steady-state throughput is 1.0 for all STM simulations with three or more cache elements, except for the eight bank cases with three and four cache elements which have a steady-state throughput of 0.89 and 0.96, respectively. The detailed simulation runs for eight banks with three and four cache elements are shown in Figure VI.64 and Figure VI.65 respectively. The three-cache-

element-configured simulation reveals the GR becoming inactive for approximately 15 cycles indicating an insufficient number of cache elements. The additional cache element in Figure VI.65 eliminates all but two of these cycles. The radix-2 set of simulations represent a situation where a few more cache elements may be useful even though they are not needed most of the time. Figure VI.63 reveals that the simulated maximum latency is consistent with constant stride upper bound except for the eight-bank cases.



**Figure VI.62 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=2 (Permutation-Based Decoding)**



**Figure VI.63 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=2 (Permutation-Based Decoding)**



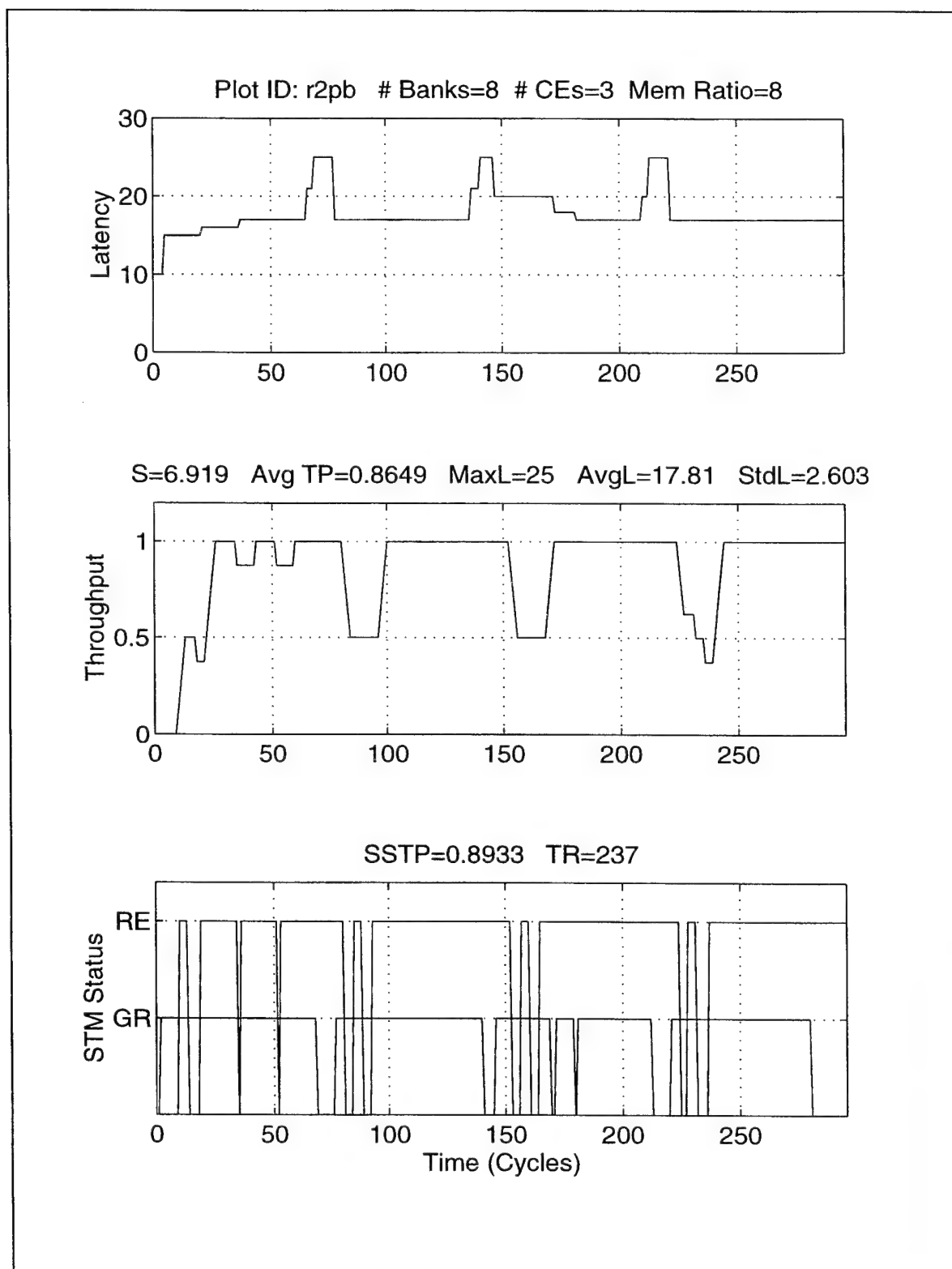


Figure VI.64 Detail Simulation Run for Radix-2 STM(8,3,8) (Permutation-Based Decoding)

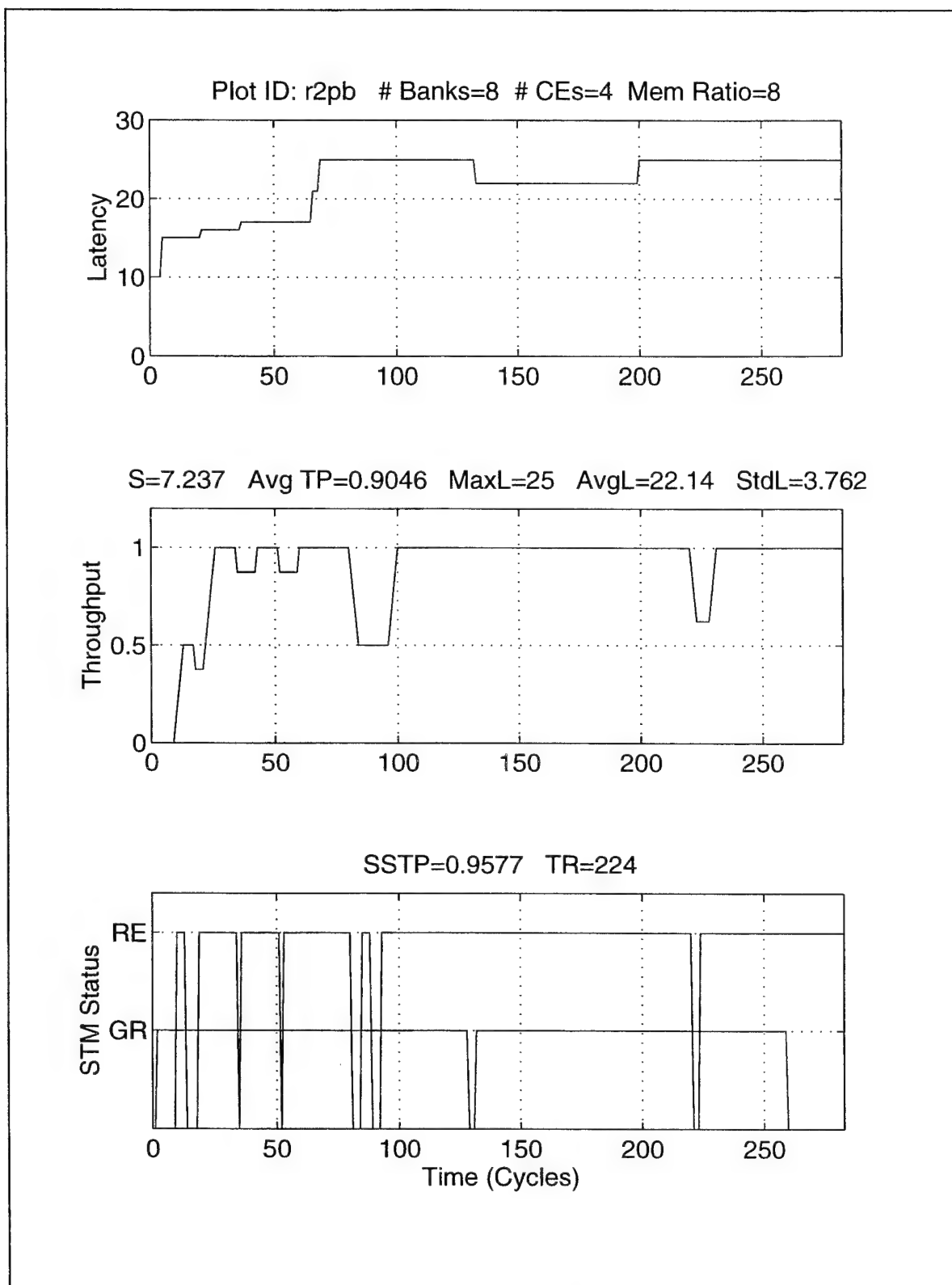
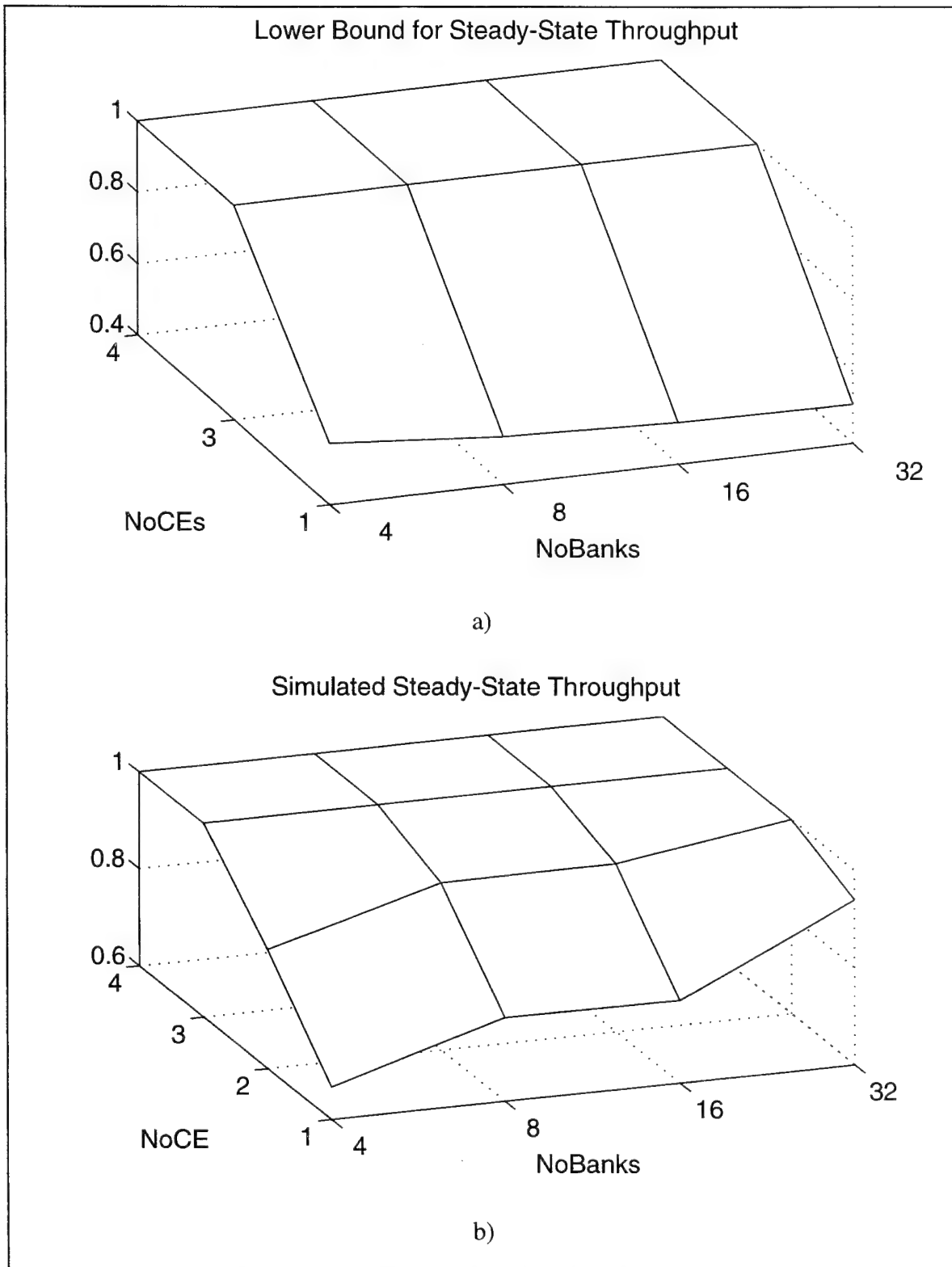


Figure VI.65 Detail Simulation Run for Radix-2 STM(8,4,8) (Permutation-Based Decoding)

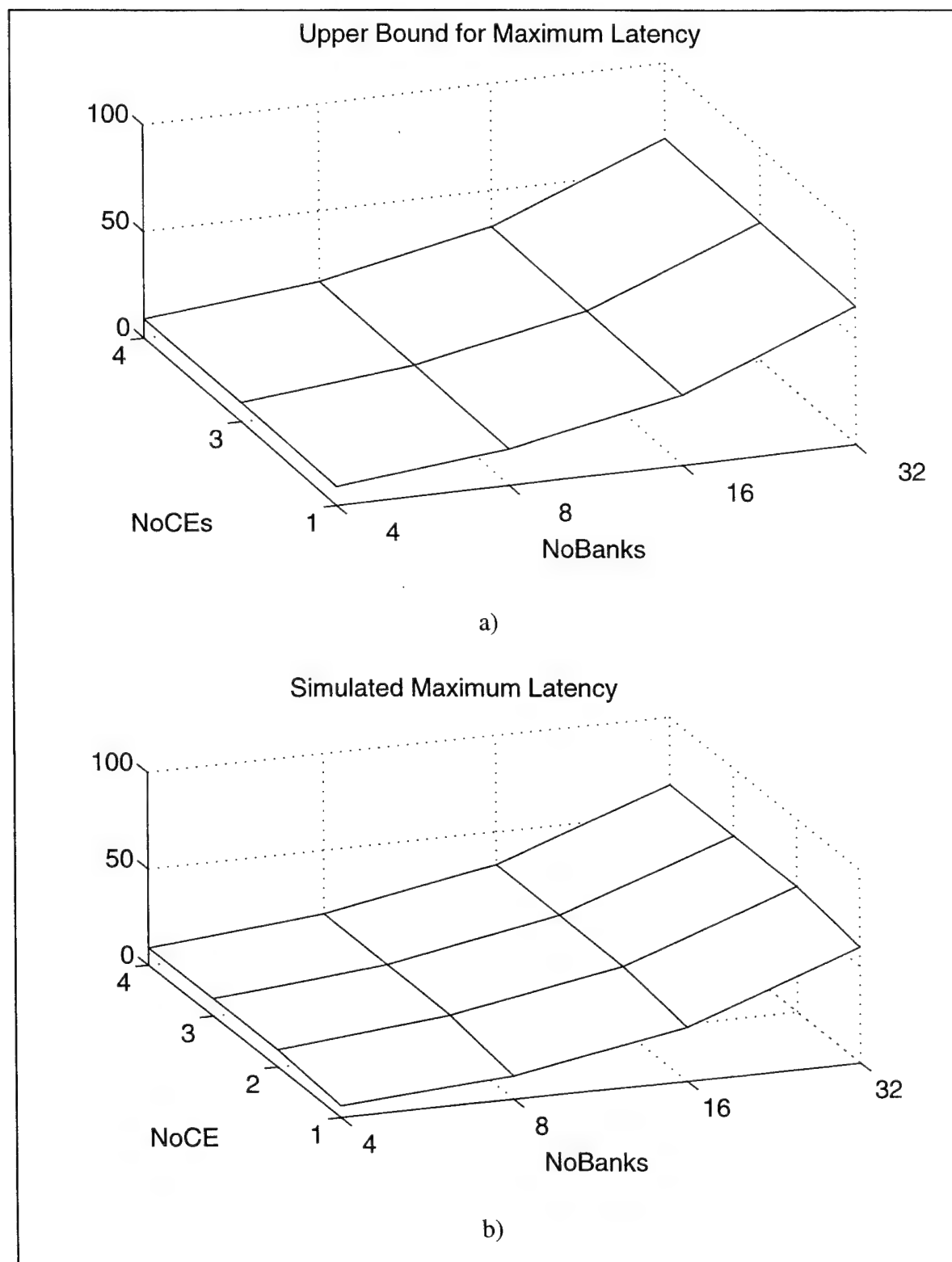
The steady-state throughput for radix four, eight and 16 shown in Figure VI.66, Figure VI.68, and Figure VI.70 reveal an ideal steady-state throughput for all STM cases where the number of cache elements is three or four, as the theory predicts. There is substantial degradation for most standard interleaving cases. A two-cache-element STM performs better for a larger number of banks and poorly for four bank scenarios. The maximum latency was equal to the theoretical upper bound in all cases, except for the 32-bank configurations for radix eight and 16 simulations as shown in Figure VI.67, Figure VI.69, and Figure VI.71.

In summary, with the aid of permutation matrices tailored to the stride between a radix butterfly operation, the resulting pattern has the features of a constant stride pattern resulting in an optimal steady-state throughput when at least three cache elements are present. Further, the maximum latency is limited to approximately twice the ideal latency for interleaved memory systems. These results apply to radices of four, eight, and 16. Radix-2 butterfly patterns were found to yield good performance although not quite as good as with the tailored matrices using a generic constant stride permutation matrix as shown in Figure III.13 through Figure III.16.

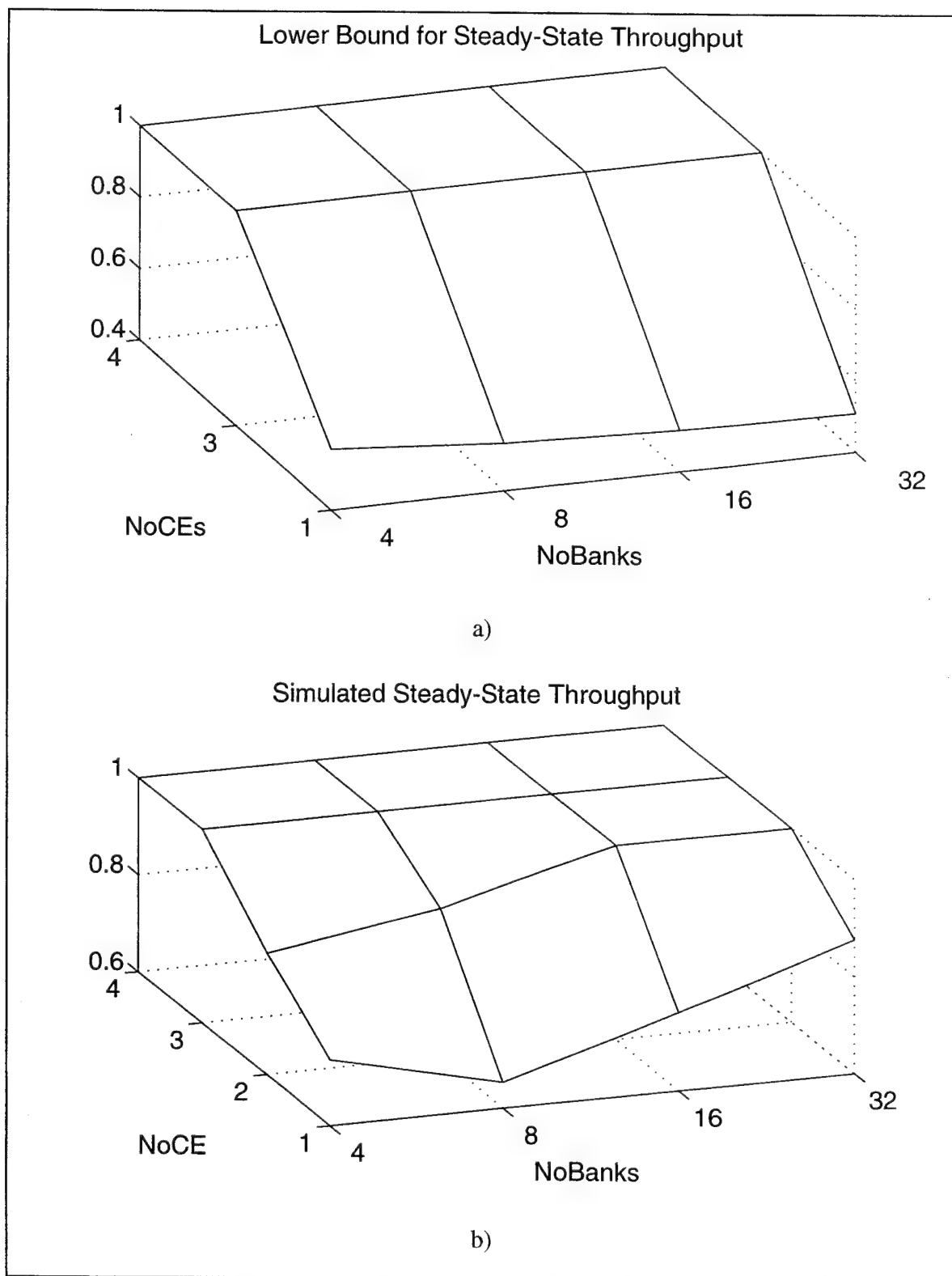
The next section will describe the performance obtained when applying conventional decoding to digit-reversed address patterns.



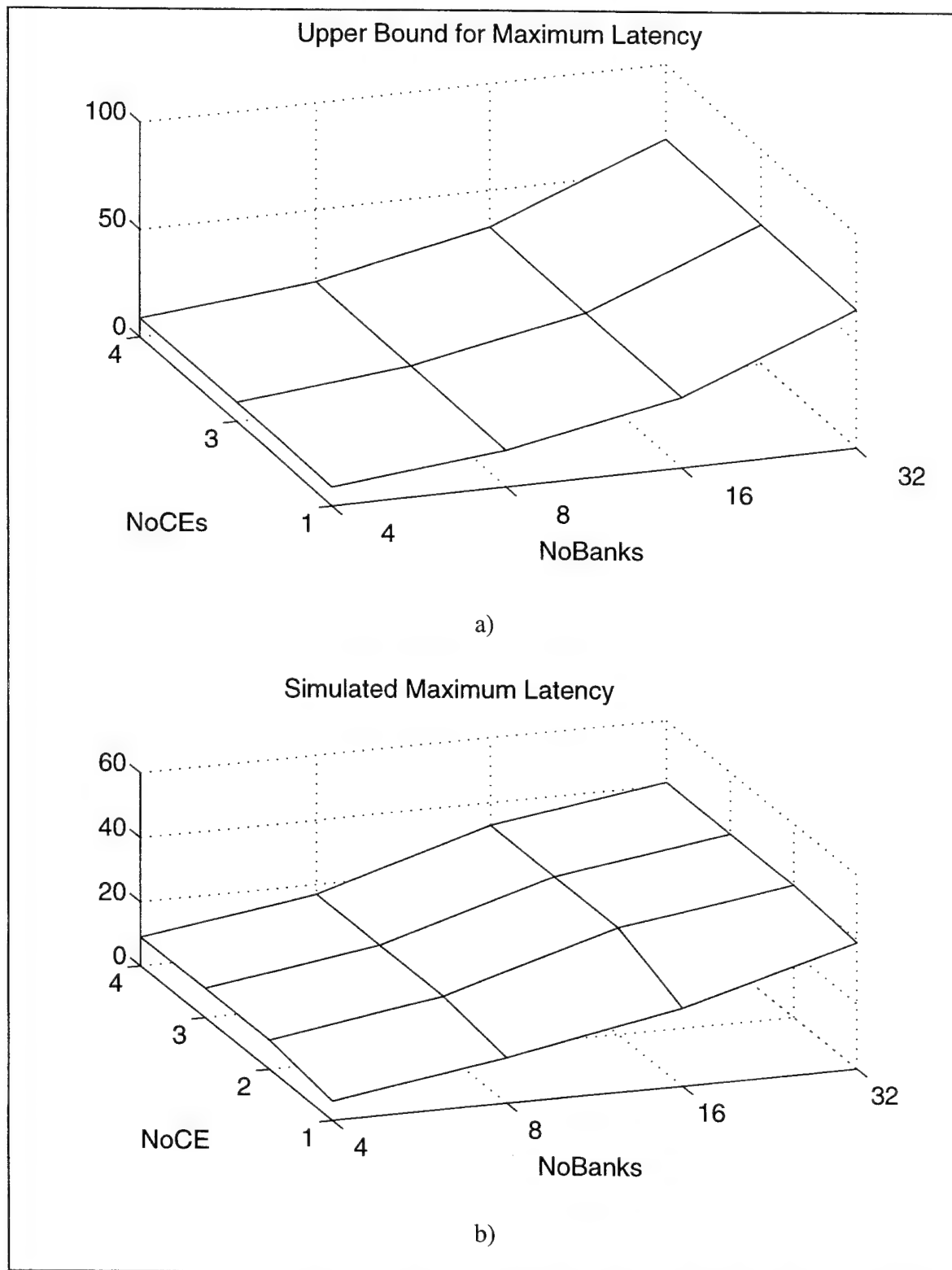
**Figure VI.66 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=4 (Permutation-Based Decoding)**



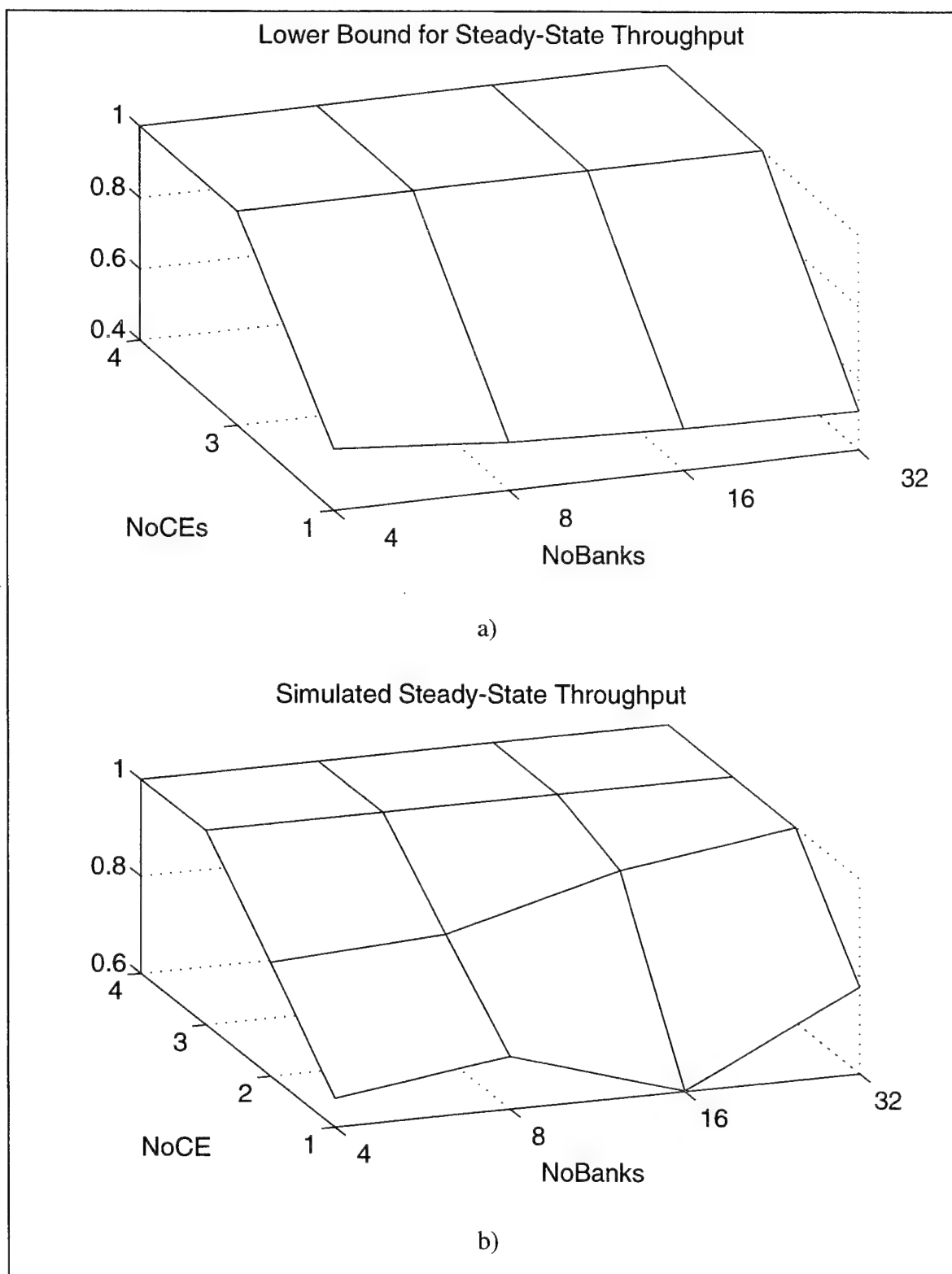
**Figure VI.67 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=4 (Permutation-Based Decoding)**



**Figure VI.68 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=8 (Permutation-Based Decoding)**

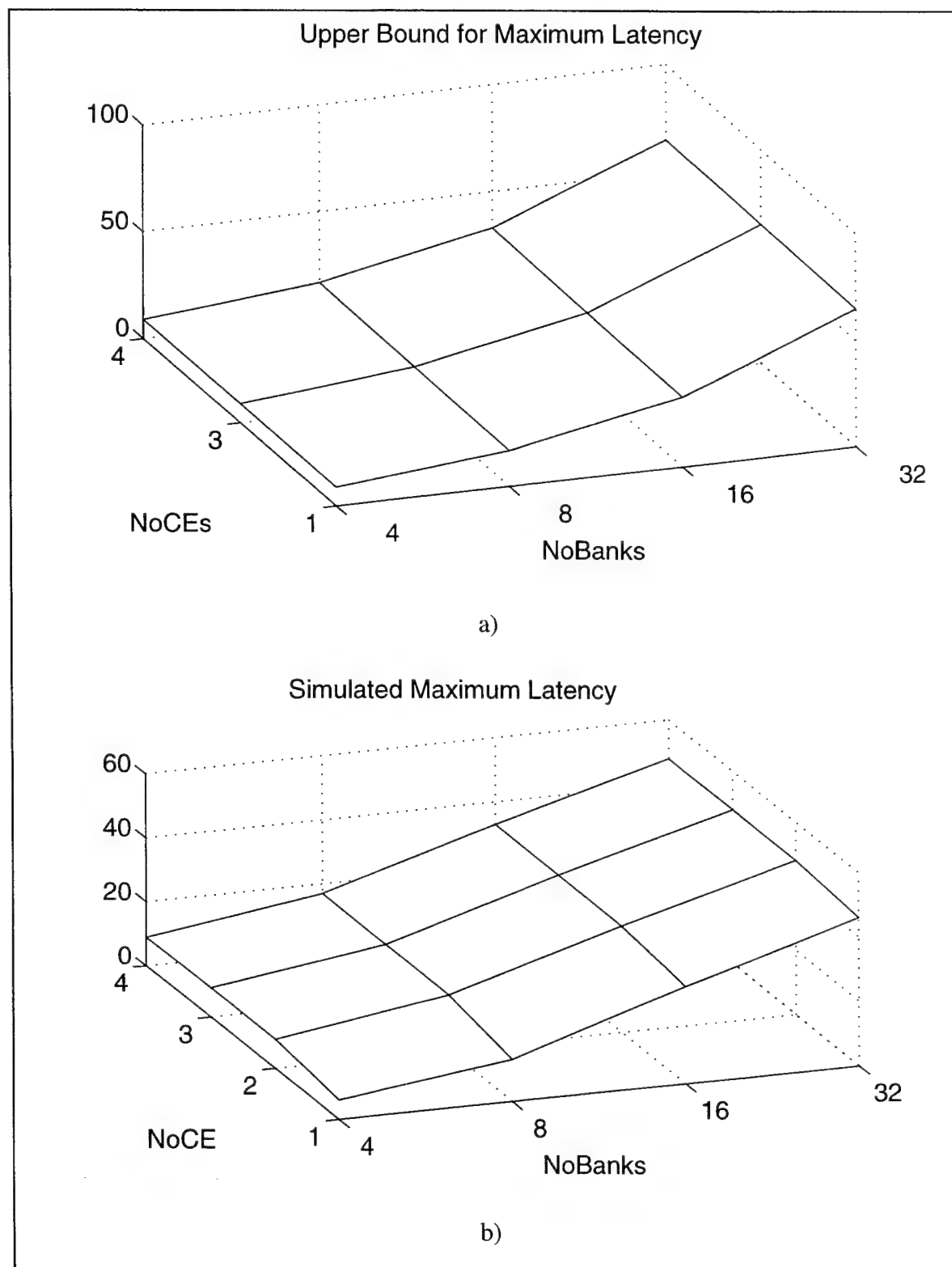


**Figure VI.69 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=8 (Permutation-Based Decoding)**



**Figure VI.70 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=16 (Permutation-Based Decoding)**





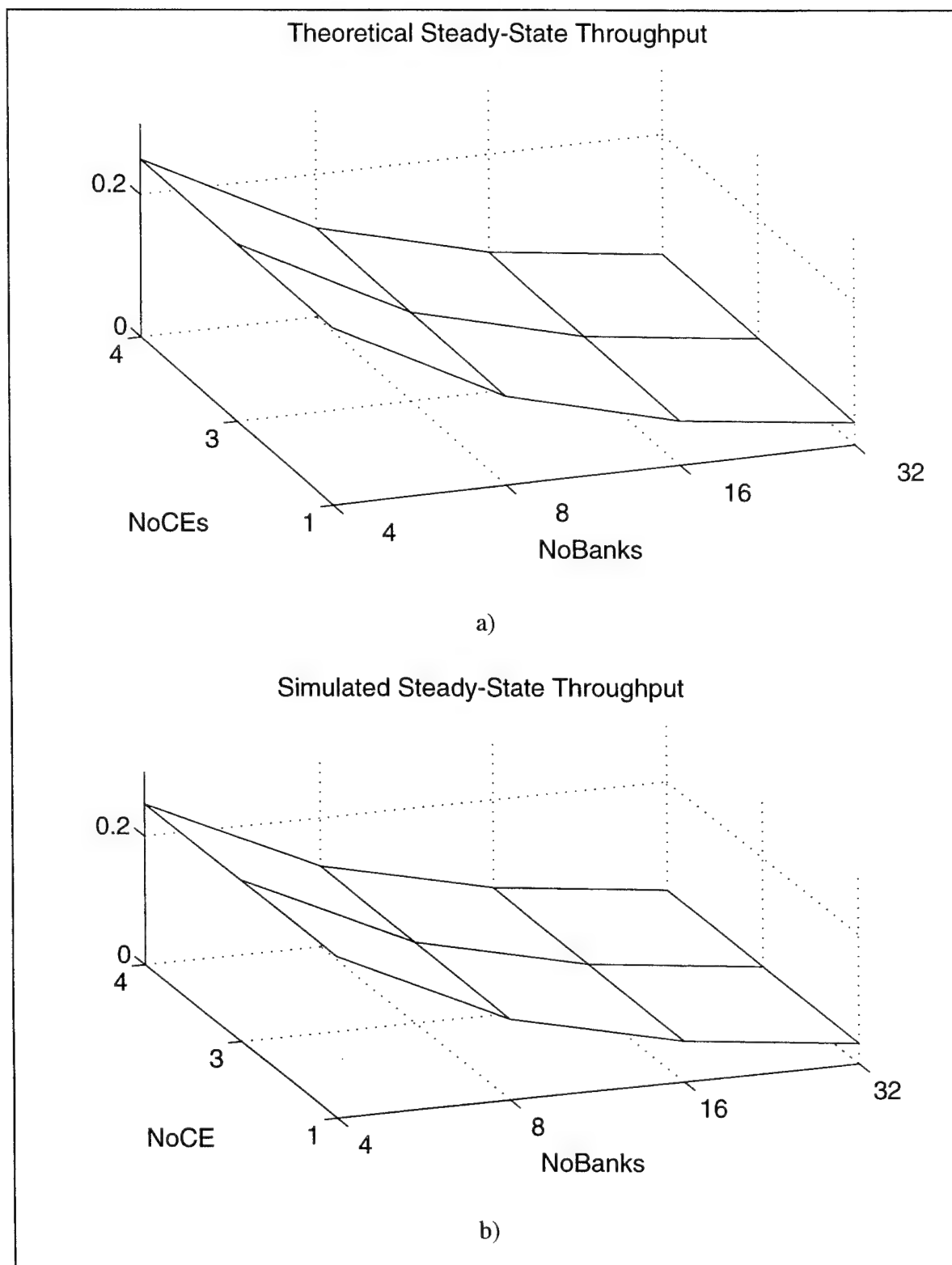
**Figure VI.71 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=16 (Permutation-Based Decoding)**

## 5. Digit Reversed: Conventional Memory Decoding

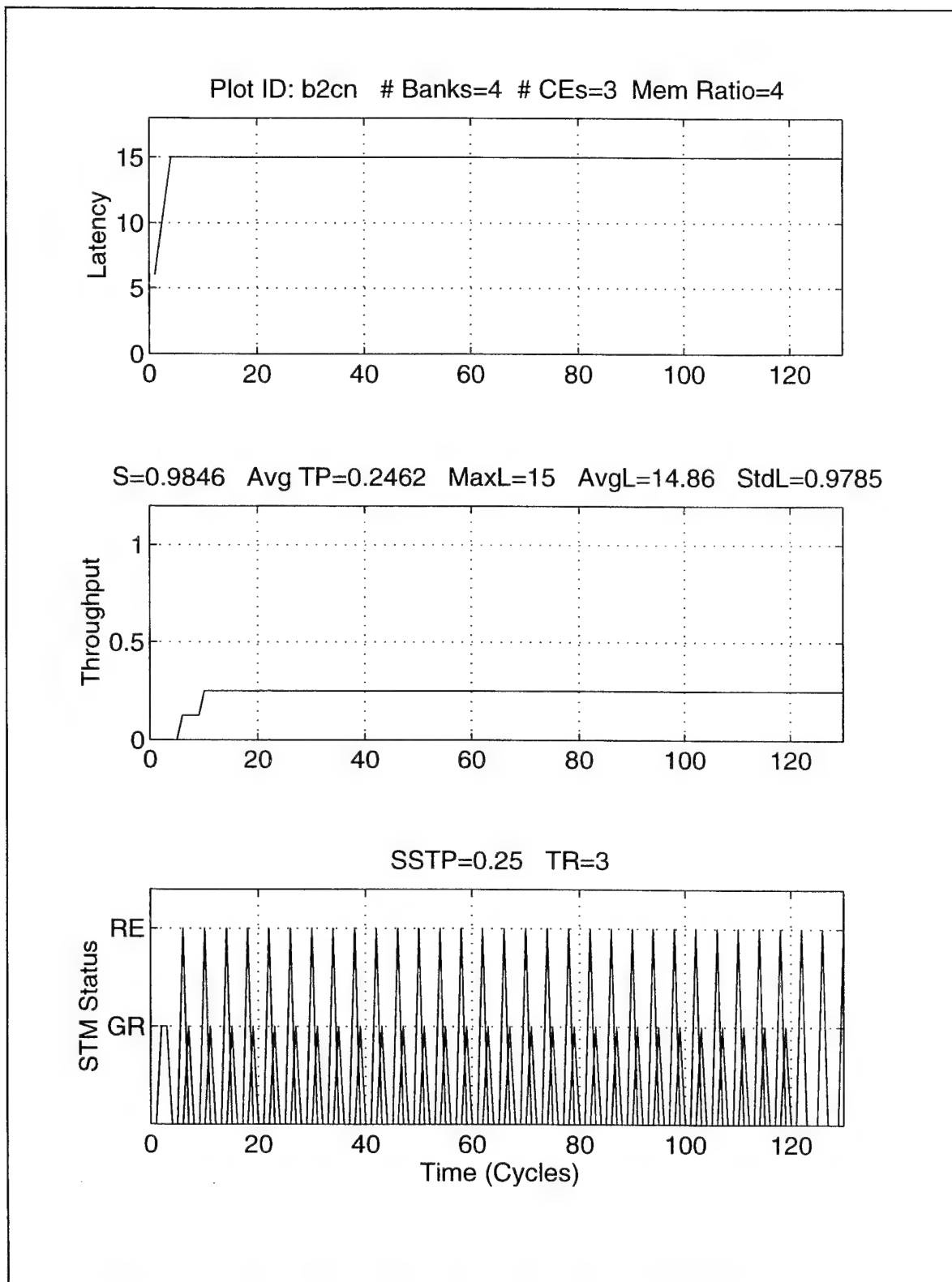
One experiment was conducted to demonstrate the performance when using conventional decoding for digit-reversed address patterns. The input stream was a digit-reversed pattern for a radix of two with ten digits, yielding a stride of  $2^9$  for the radix operation. A comparison of the theoretical versus simulated steady-state throughput is shown in Figure VI.72. The theoretical results matches the simulated results perfectly. A steady-state throughput is obtained that is the reciprocal of the number of banks and is independent of the number of cache elements.

A detailed simulation for a memory with four banks and three cache elements is shown in Figure VI.73. An examination of the STM Status plot indicates that the RE line is active once every four cycles yielding a throughput of 0.25. Note that the GR signal is active for a short period of time allowing the one active bank's cache elements to be filled with requests. Thereafter, the RE line is active filling the one available cache element followed by processing time for a memory request and then an output signaled by an active RE line. This pattern is repeated until the simulation is completed. Figure VI.74 contains a similar plot for a memory system with 32 banks. The primary difference is that the active RE lines are separated by 32 cycles rather than four as in Figure VI.73 because the memory ratios are matched to the number of banks. The resulting throughput is  $1/32$  or approximately 0.03125 as indicated on the figure.

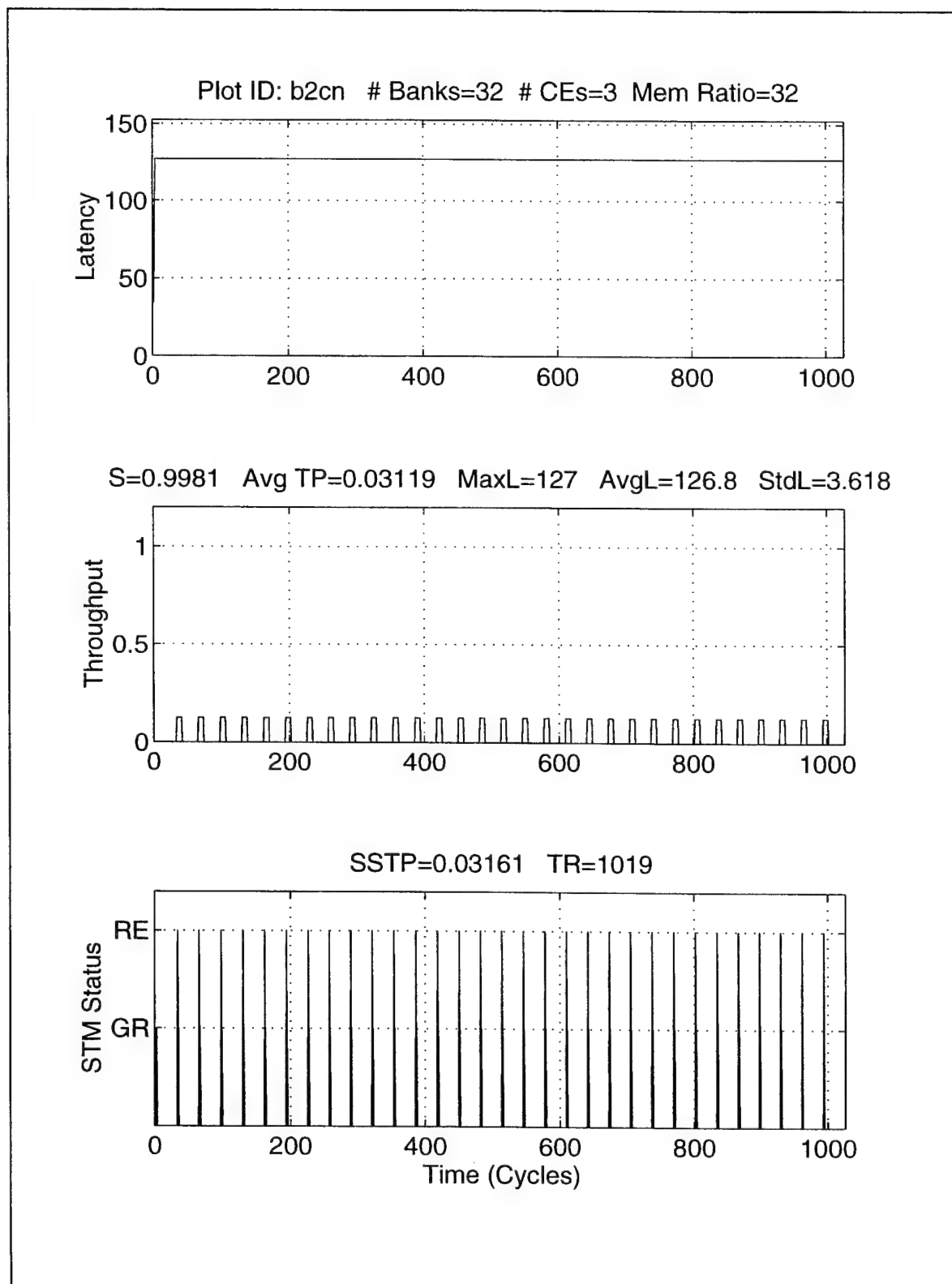
This result presents a major obstacle to the architecture described in Chapter 0 because one such pass is needed for each FFT. The next section describes the results obtained when permutation-based decoding is used for digit-reversed address patterns.



**Figure VI.72 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=2 / NoDigits=10 (Conventional Decoding)**



**Figure VI.73 Detail Simulation Run for Radix=2 / NoDigits=10 STM(4,3,4)  
(Conventional Decoding)**



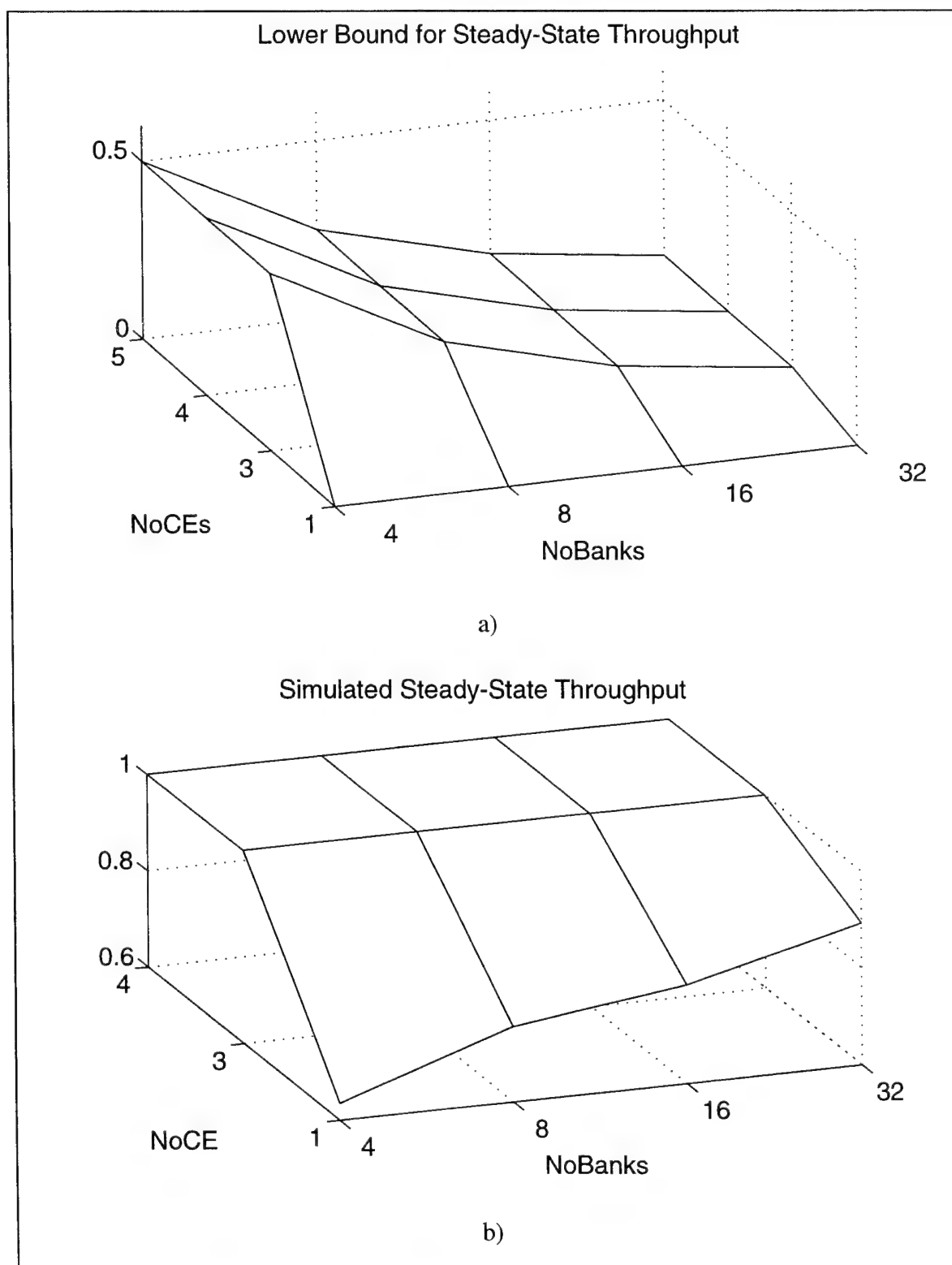
**Figure VI.74 Detail Simulation Run for Radix=2 / NoDigits=10 STM(32,3,32)  
(Conventional Decoding)**

## **6. Digit Reversed: Permutation-Based Memory Decoding**

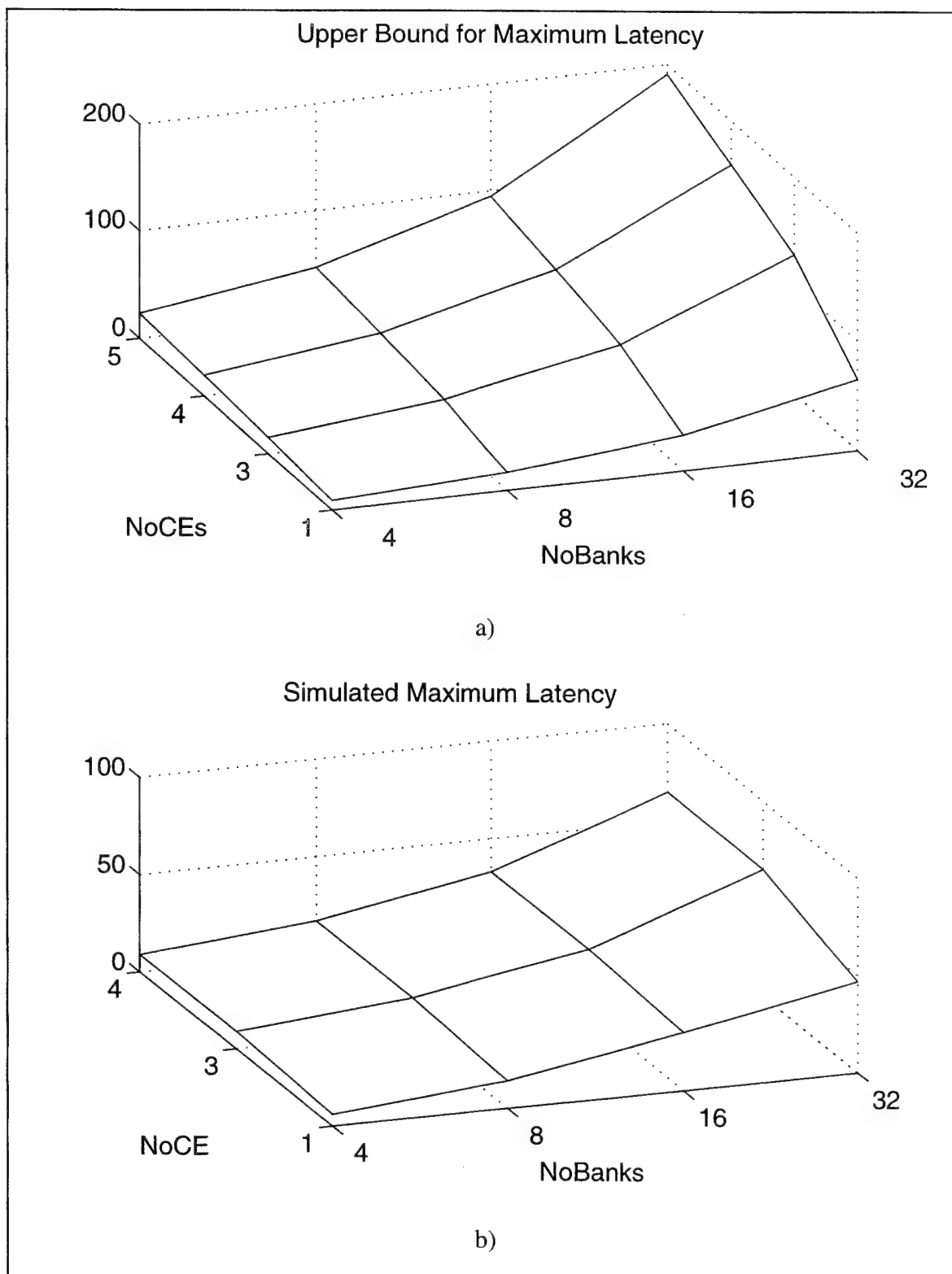
As indicated in Chapter V, the digit reversal pattern should be equivalent to constant-stride address performance if the radix is equal to or greater than the number of banks. In the case of radix-2 with ten digit simulation, the condition is not met for any of the simulations. In spite of this, the performance in this instance is almost perfect as can be seen by viewing the steady-state throughput and the maximum latencies contained in Figure VI.75 and Figure VI.76.

Figure VI.77, Figure VI.78, and Figure VI.79 contain the steady-state throughput plots for radix-4, 8 and 16 respectively. For each plot, when the radix is equal to or greater than the number of banks, an optimal steady-state throughput is obtained, as predicted in Chapter V (i.e., for four banks in Figure VI.77, four and eight banks in Figure VI.78, and four, eight, and sixteen banks in Figure VI.79). When the condition is not met, good performance is sometimes obtained anyway (e.g., 16 banks in Figure VI.77). In some instances poor performance is improved substantially by adding another cache element (e.g., eight banks in Figure VI.77 and 32 banks in Figure VI.78).

In summary, the permutation-based digit-reversed simulation results confirmed the analysis described in Chapter V. In particular, when the radix is equal to or greater than the number of banks, performance is consistent with constant-stride address patterns. When it is not, the performance is mixed but over all it provides performance that may be acceptable, given this address pass is required only once for each FFT.

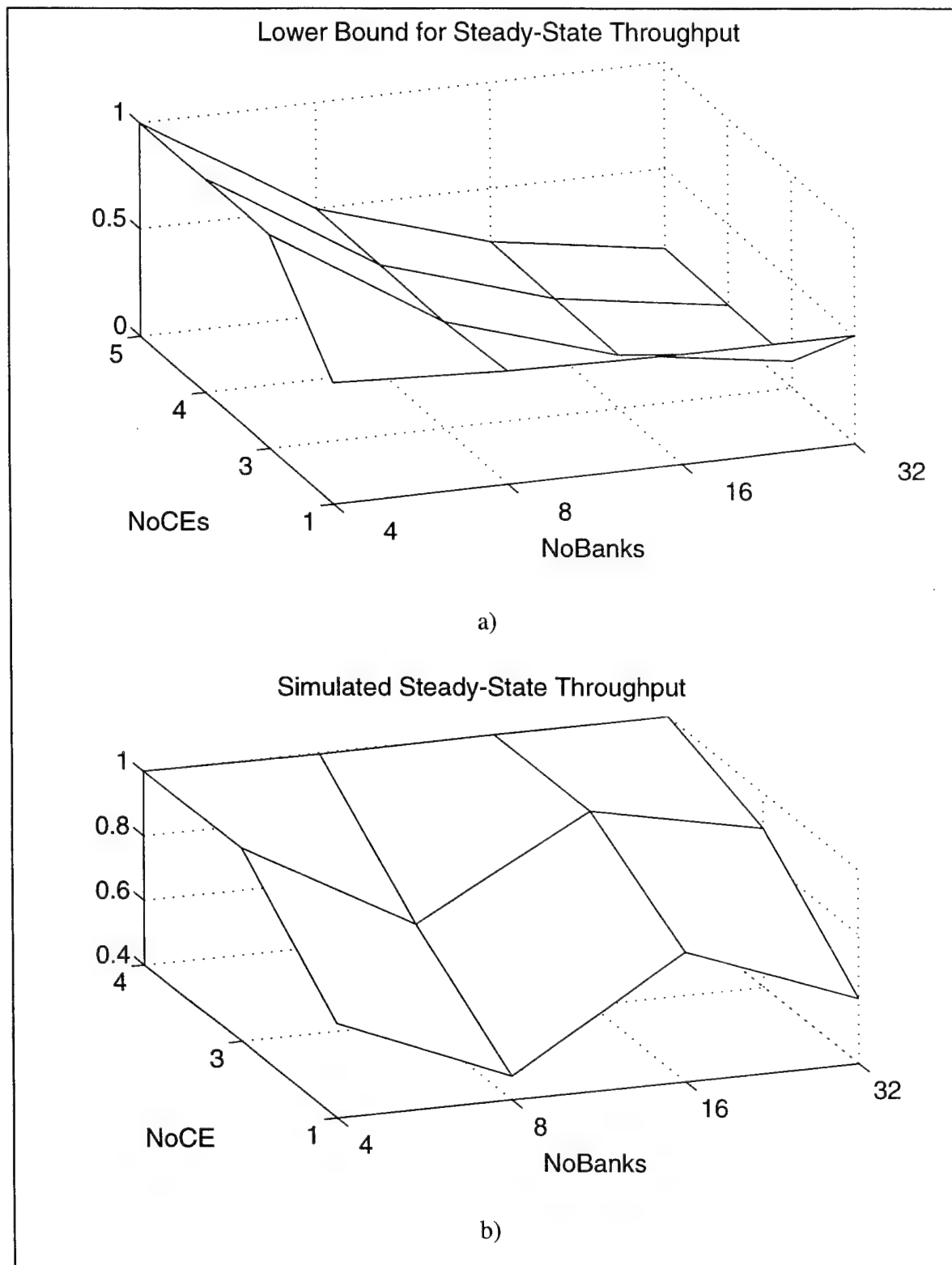


**Figure VI.75 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=2 / NoDigits=10 (Permutation-Based Decoding)**

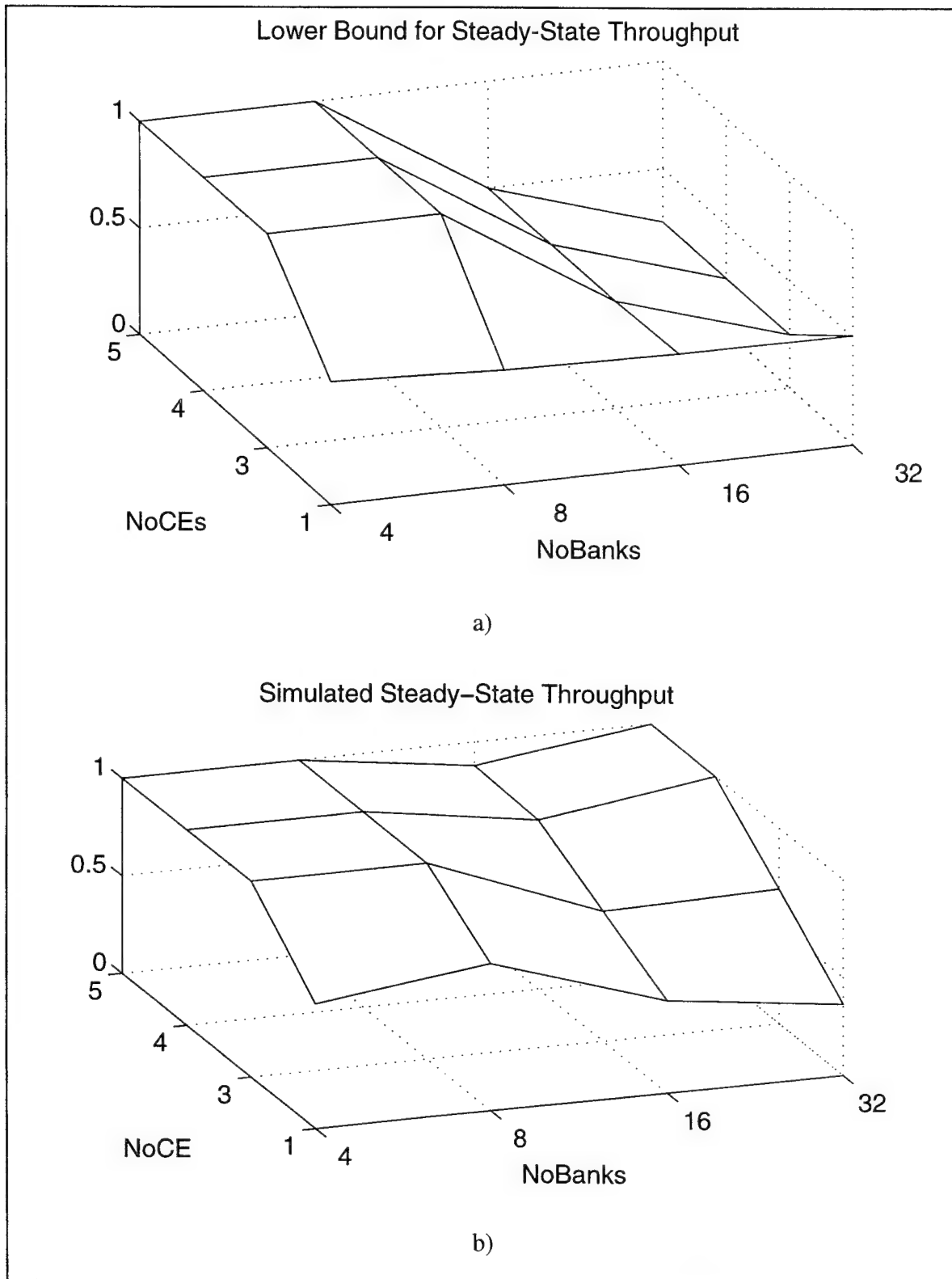


**Figure VI.76 Comparison of Theoretical Versus Simulated Maximum Latency for Radix=2 / NoDigits=10 (Permutation-Based Decoding)**

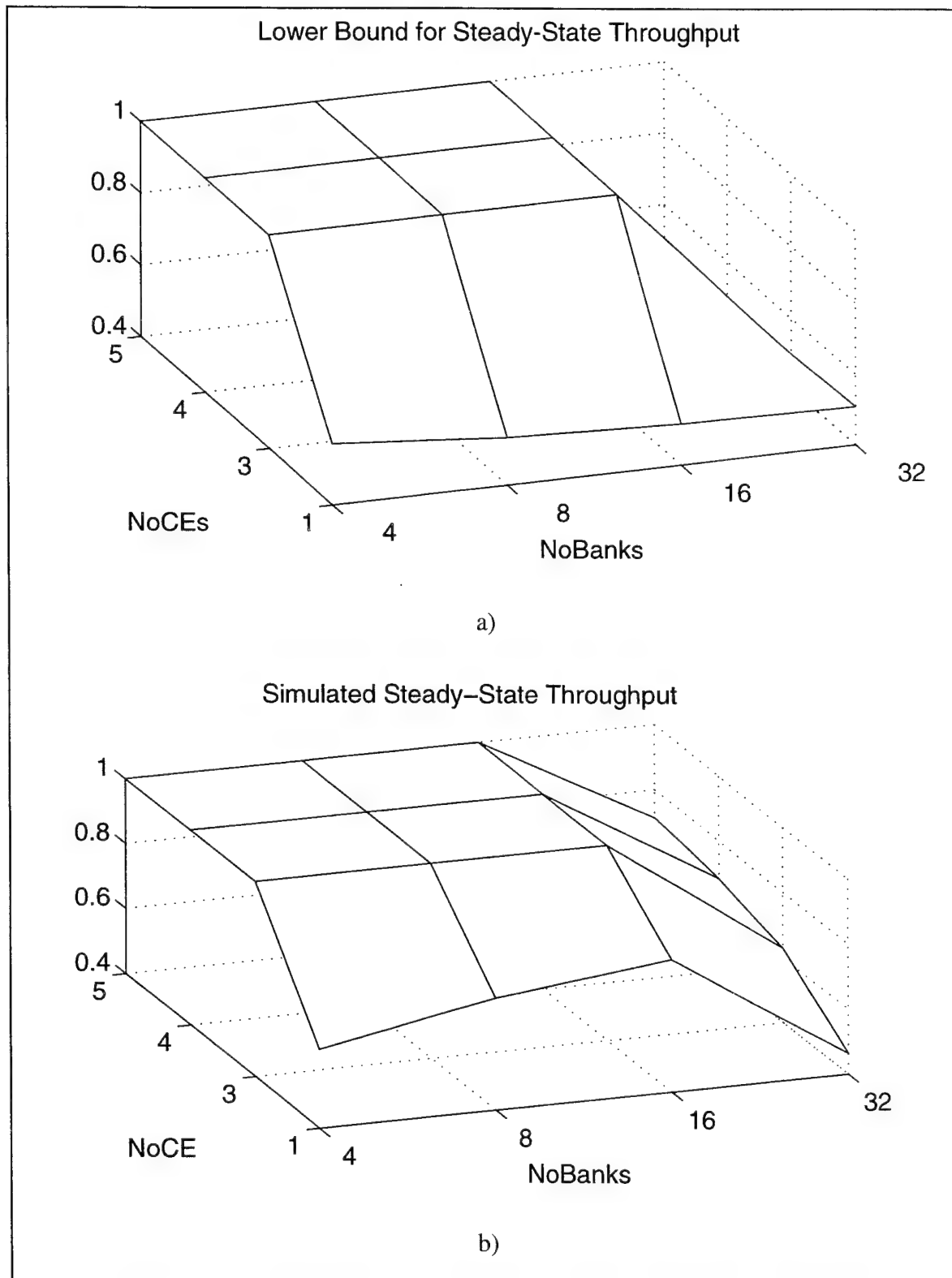




**Figure VI.77 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=4 / NoDigits=5 (Permutation-Based Decoding)**



**Figure VI.78 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=8 / NoDigits=4 (Permutation-Based Decoding)**



**Figure VI.79 Comparison of Theoretical Versus Simulated Steady-State Throughput for Radix=16 / NoDigits=3 (Permutation-Based Decoding)**

### C. GENERAL-PURPOSE COMPUTING EXPERIMENT

The speedup, throughput, and latency plots for the general-purpose computer experiment are shown in Figure VI.80 through Figure VI.82. Adding cache elements increases both the speedup and throughput. However, this is accompanied by much larger latencies for the simulation runs with a larger number of banks. For four banks, speedup increases by 382 percent from standard interleaving simulation to the STM simulation with 64 cache elements. However, 294 percent of this improvement was obtained when the number of cache elements was increased to only four. The 64-bank simulations recorded a similar trend with 406 percent total improvement and 241 percent obtained with four cache elements from the standard interleaving case.

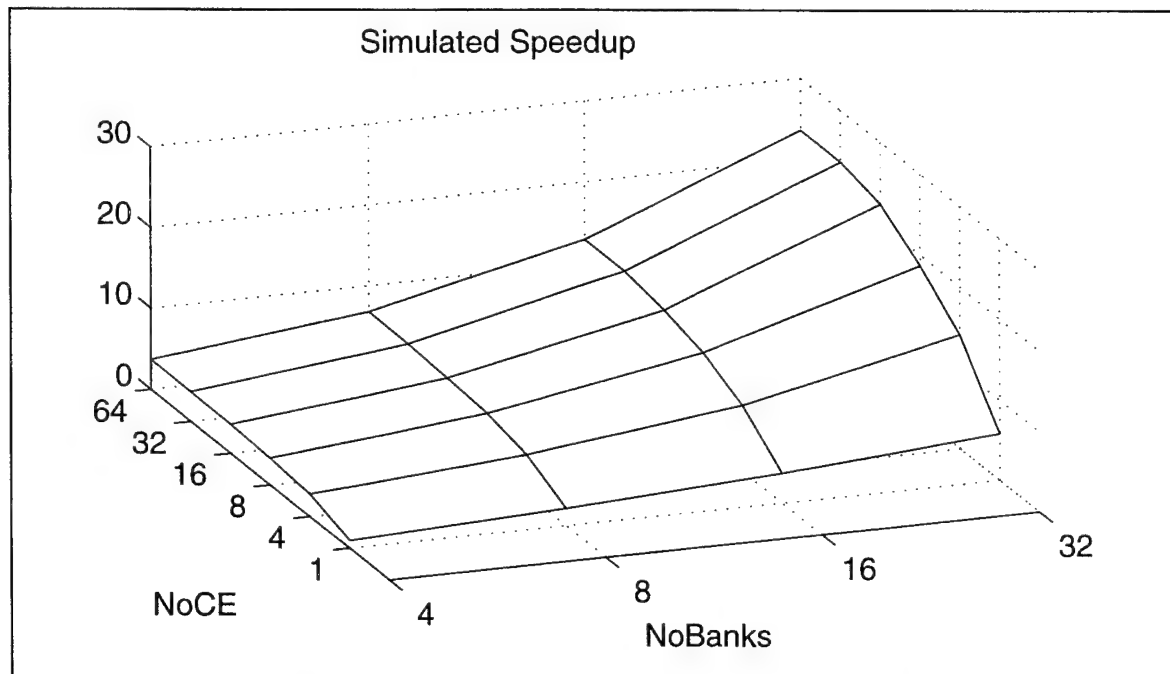
Notice that although the speedup continues to improve when cache elements and the number of banks are increased, the throughput actually falls as the number of banks increases. This is because the memory ratio is matched to the number of banks and the difficulty of the problem increases proportionally as the number of banks increases. Recall the relationship between speedup, throughput, and the memory ratio as shown in Equation (II.7).

A comparison of the standard interleaving case to the analytical results is shown in Table VI.5. Although the simulated results correlate with the analytic results, there is a constant bias of approximately one for each value.

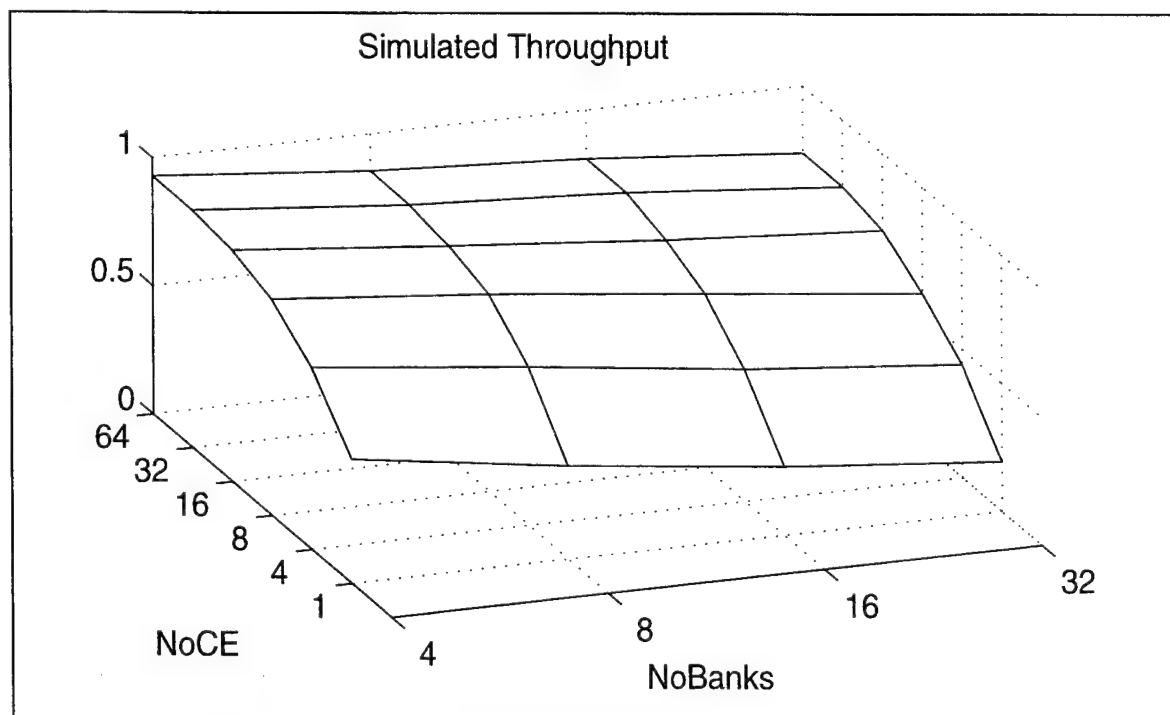
In summary, the general-purpose computing simulation suggests that speedup and throughput are enhanced by adding cache elements. Diminishing marginal returns is observed when only a few cache elements are added to the standard interleaving case.

Number of Banks	$B^{0.56}$	Simulated Results
4	2.2	1.2
8	3.2	2.2
16	4.7	3.7
32	7.0	5.8

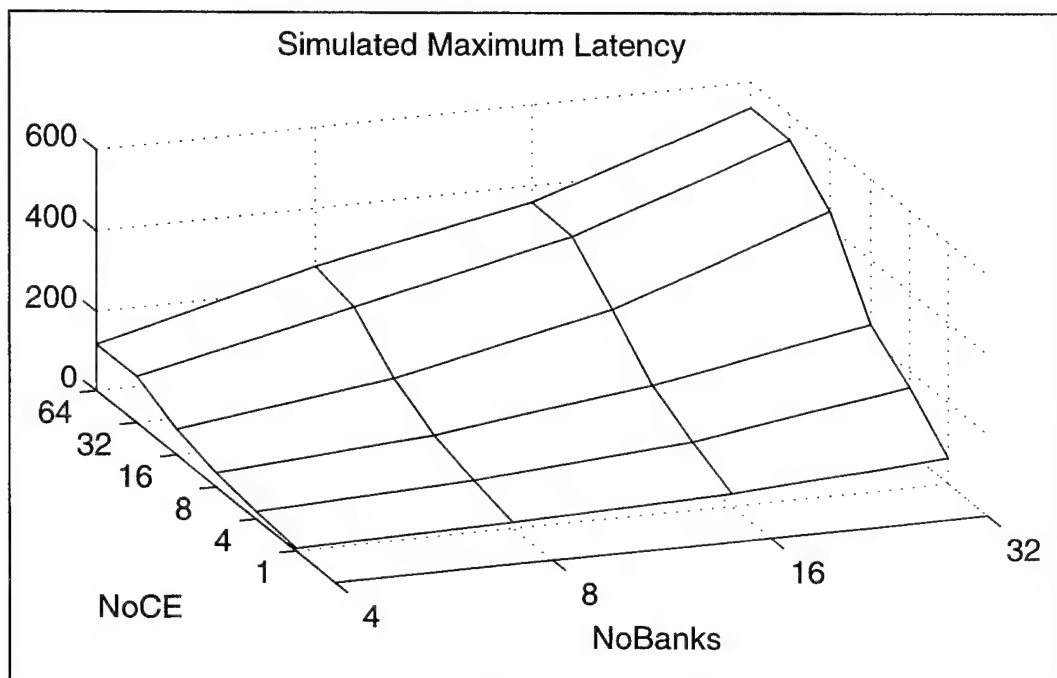
**Table VI.5 Comparison of Analytic Versus Simulated Speedup**



**Figure VI.80 General-Purpose Experiment: Speedup**



**Figure VI.81 General-Purpose Experiment: Throughput**



**Figure VI.82 General-Purpose Experiment: Maximum Latency**

## VII. CONCLUSIONS

This section first summarizes the design decisions for the Butterfly Machine (BFM) Architecture followed by a design methodology for constructing this type of computer. The last section contains additional conclusions concerning this effort.

### A. DESIGN DECISIONS

The computational complexity of cyclostationary processing, combined with the requirement to design to a factor of real time  $F_T$  and sample interval  $T_s$ , drives the need for a scaleable number of processors in the architecture. The BFM Architecture is based on pipelined vector processing techniques because it yields an efficient implementation for FFTs in particular, and vector operations in general.

Radix- $2^k$  algorithms were selected based on the availability of efficient implementations in hardware and their widespread use and popularity. The radix values supported are two, four, eight, and sixteen.

The number of memory banks allowed in the architecture is a power of two. This constraint simplifies the bank number selection hardware although  $2^k \pm 1$  bank architectures are almost competitive.

The use of radix values and number of banks that are both powers of two require an alternative to conventional bank number decoding. Properly designed permutation-based bank decoding provides an efficient utilization of the memory.

Another design decision is programmable permutation matrices. This provides for flexibility in general. The primary motivation is to enhance performance by allowing radix- $r$  specific matrices. This design decision is closely related to two additional design decisions:

- The decision not to use a specialized permutation matrix for the radix-2 butterfly.
- The decision to require that all specialized radix- $r$  butterfly matrices also support constant strides of powers of two.

The computation of an FFT on an input vector of length  $2^k$  requires that a decision be made concerning the size of the radices and order in which the different radix operations are applied. The strategy taken for this architecture is to rewrite the length of the input vector as

$$2^k = R^m \cdot \bar{R} \quad (\text{VII.1})$$

where  $R$  is the largest valid radix for the length of the input vector. The largest value of  $m$  is chosen such that

$$\begin{aligned} 2^k &\geq R^m \text{ and} \\ R &> \bar{R}. \end{aligned} \quad (\text{VII.2})$$

Therefore,  $m$  radix- $R$  butterfly passes will be made on the input vector followed by at most one radix- $\bar{R}$  butterfly pass.

An inspection of Figure III.9 reveals that the memory that initially holds the input vector must be accessed by an address pattern of constant stride of one followed by a radix- $R$  pattern. Therefore, this memory will be loaded with the appropriate radix- $R$  permutation matrix. Each additional pass is characterized by a memory that will accept a set of inputs with a constant stride of one followed by read operation with a radix- $r$  butterfly address pattern. The corresponding memory will use the appropriate radix- $r$  permutation matrix. The appropriate matrix is the radix- $R$  permutation matrix for all passes with the possible exception of the last pass which will use the radix- $\bar{R}$  permutation matrix if  $\bar{R}$  exists for the decomposition of Equation (VII.1).

The right-most memory in Figure (III.9) is written into with a constant stride of one. The data is read out with a digit-reversed pattern. In those cases where the vector length is such that the decomposition of Equation (VII.1) does not contain the factor  $\bar{R}$ , then the required addressing pattern is a digit-reversed address pattern. If on the other hand, there is a factor of  $\bar{R}$  in Equation (VII.1), then the address pattern is not strictly digit-reversed. However, the address pattern does have a characteristic of a constant stride of a power of two. In either case, if the radix is greater than the number of banks, then the performance is near optimum.



When the radix is less than the number of bank, the simulation results suggest that the steady-state throughput is near optimal when four cache elements are used for those cases examined in Chapter VI (See Figure VI.77 and Figure VI.78). These address patterns must be simulated in any final design to verify performance and to adjust the permutation matrices if the performance is not acceptable.

The permutation matrices used for the digit-reversed pattern experiments were the constant stride powers of two matrices. A future research topic is to determine whether a tailored permutation matrix can be found for the digit reversed case.

## B. STM DESIGN METHODOLOGY

The design methodology for developing an STM memory begins with the processor and bulk store memory cycle times desired for the architecture. The ratio of the bulk store cycle time to the processor cycle time is the memory ratio, one of the three parameters necessary for STM memory. The memory ratio can be expressed as

$$MR = \left\lceil \frac{T_{bs}}{T_{pr}} \right\rceil \quad (\text{VII.3})$$

where

$T_{bs}$  is the cycle time for the bulk store, and

$T_{pr}$  is the cycle time for the processor.

The ceiling function must be taken on the bulk store / processor cycle time ratio to yield an integer value that will permit the memory to process a memory request.

The memory ratio dictates the number of banks required for the memory system. The number of banks is required to be a power of two in this design for simplicity of bank selection and must be greater than or equal to the memory ratio.

The results in Chapter VI suggest that overall performance is constrained by the maximum latency and the maximum latency is approximately twice the memory ratio when permutation matrices are utilized for bank decoding. Since the memory ratio is tied directly to the number of banks, there is motivation to minimize the number of banks.

Once the number of banks has been selected, the last parameter to fix is the number of cache elements. Based on the results of Chapter VI, the number of cache elements is likely to be not less than four. However, cache elements are relatively inexpensive, assuming that they are implemented with very large scale integration. The actual number chosen is likely the largest number possible within the economic bounds of the fabrication process.

Programmable permutation matrices allow the incorporation of performance enhancements when more advanced permutation matrices are discovered. In some circumstances, the performance of these matrices may be dependent upon more cache elements than was previously required.

The last step of the design process is to construct permutation matrices for the architecture. Although there are many possible addressing patterns, there are a relatively small number when compared to general-purpose computing. All, or a selected set, can be simulated to verify the anticipated performance. The number of cache elements can be varied for sensitivity analysis. Permutation matrices may also be fine tuned to improve performance.

## **C. GENERAL CONCLUSIONS**

The preceding chapters describe a pipelined vector computer architecture designed to compute fast Fourier transforms (FFTs) efficiently. Other vector processing operations such as vector multiplication are also well suited for this architecture. Use of the constant geometry radix butterfly organization is a key design decision providing simplification in the address stream generation for radix- $r$  passes.

The memory system is the key component of a vector processor architecture. Addressing stream characteristics for general-purpose and vector processors are described in Chapter II. Banked interleaved memory remains the technique of choice for vector processors because of the high-performance requirements and the promise of exploiting the constant-stride address stream characteristic. This architecture is based on the requirement that data be fed into a vector processor at the rate of one data element per clock cycle for each vector. The constant-stride address stream characteristic is exploited

through the use of specially designed permutation matrices used for bank number decoding.

The performance of STM memories using both conventional and permutation-based matrices was analyzed in Chapter V and Chapter VI. The preferred bank decoding scheme was with permutation matrices, based on performance. The results of Chapter VI indicate that optimum steady-state throughput is possible in all cases for constant-stride address patterns with a stride that is a power of two, as well as for radix- $r$  butterfly patterns using tailored permutation matrices. In fact, both of these cases yield an upper bound of twice the memory ratio plus one. This is excellent given that the minimum latency for any interleaved system is the memory ratio plus two! The other address pattern, digit-reversed addressing, also yields the same performance as indicated above for constant-stride and radix- $r$  butterfly addressing when the radix is greater than or equal to the number of banks. When it is not, the actual performance is in some instances similar to that noted above, and in others is somewhat less. These cases need to be simulated to determine the specific performance characteristics. One possible area of study is to determine whether permutation matrices can be designed specifically for digit-reversed patterns and still retain their suitability for constant stride and radix- $r$  address patterns.

The following is a list of further conclusions concerning the butterfly machine architecture and the STM memory described previously:

- The use of BFM's provide a practical method for reducing the clock time needed for cyclostationary computing. The amount of reduction is variable and is determined by the degree to which parallelism is exploited.
- The use of BFM's is scaleable over a substantial processing range and is limited by the number of backplane slots supported by the host. An architecture using a single BFM chip is first described which provides a baseline capability. Due to the parallelism inherent in many cyclostationary algorithms, a natural extension is to develop an architecture that incorporates

multiple copies of the one-chip architecture and connect them with dedicated high-speed data busses for data sharing.

- The BFM architecture requires large quantities of memory. This memory requirement can be accommodated using relatively slow low-cost bulk memory devices. In this investigation, each addressing stream had a dedicated memory. One area of future study is to determine if it is more effective to construct fewer larger memories than the configuration shown in Figure III.17.
- A good design requires that the number of banks be greater than or equal to the memory ratio. With the appropriate permutation matrix, the number of banks need not be greater than the memory ratio. Further, the number of cache elements can be limited to approximately four in most circumstances.
- STM is an effective technique for using relatively slow, inexpensive, bulk storage with the BFM architecture when the array lengths are large, relative to the latency.

## LIST OF REFERENCES

---

1. Gardner, W. A., *Statistical Spectral Analysis: A Nonprobabilistic Theory*, Englewood Cliffs, NJ: Prentice-Hall, 1987.
2. Brown, W. A., and Loomis H. H., Jr., "Digital Implementations of Spectral Correlation Analyzers," *IEEE Trans. On Signal Processing*, Vol. 41, pp. 703-720, February 1993.
3. Roberts, R. S., Brown W. A., Loomis H. H., Jr., "A Review of Digital Spectral Correlation Analysis: Theory and Implementation" Article 6 of *Cyclostationary in Communications and Signal Processing*, Piscataway, NJ, IEEE Press, 1994.
4. Gardner, W. A., *Cyclostationary in Communications and Signal Processing*, Piscataway, NJ, IEEE Press, 1994.
5. Roberts, R. S., "Architectures for Digital Cyclic Spectral Analysis", Ph.D. Dissertation University of California, Davis, September 1989.
6. Roberts, R. S., Loomis, H. H., Jr., "Parallel Computation Structures for a Class of Cyclic Spectral Analysis Algorithms", *Journal of VLSI Signal Processing*, pp. 25-40, October 1995.
7. BBN Advanced Computers Inc., *TC2000 Technical Product Summary*, November 1989.
8. Pease, M. C., "Organization of Large Scale Fourier Processors," *Journal of ACM*, Vol.16, No. 3, pp. 474-482, July 1969.
9. Groginsky, H. L., and Works, G. A., "A Pipeline Fast Fourier Transform," *IEEE Transactions on Computers*, Vol. C-19, pp. 1015-1019, November 1970.
10. Filip, A. E., Frankovich, J. M., Purdy, R. J., and Blankenship, P. E., *Digital Signal Processor Designs for Radar Applications, Vol. 1 and 2*, MIT-LIN-TN-1974-58, Lincoln Laboratory, 1974.
11. Dieffenderfer, J. W., Hancke, P. J., and Schoenfeld, R. F., "Pipeline Fast Fourier Transform Processor," *IBM Technical Disclosure Bulletin*, Vol. 16, No. 2, July 1973.
12. Corinthios, M. J., Smith, K. C., and Yen, J. L., "A Parallel Radix-4 Fast Fourier Transform Computer," *IEEE Transactions on Computers*, Vol. C-24, pp. 80-92, January 1975.
13. Sapiecha, K., and Jarocki, R., "Modular Architecture for High Performance Implementation of the FRR Algorithm," *IEEE Transactions on Computers*, Vol. C-39, pp. 1464-1468, December 1990.
14. Sommer, R. E., and Mehalic, M. A., "Design Enhancements for the Air Force Institute of Technology's Winograd Fourier Transform Processor," *Proceedings of the IEEE 1991 National Aerospace and Electronics Conference NAECON 1991*, Vol. 1, pp. 90-97, 1991.
15. Franceschetti, G., Mazzeo, A., Mazzocca, N., Pascazio, V., and Schirinzi, G., "An Efficient SAR Parallel Processor," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 27, Issue 2, pp. 343-353, March 1991.
16. Comerford, R., and Watson, G. F., "Memory Catches Up," *IEEE Spectrum*, pp. 34-35, October 1992.

- 
17. Hennessy, J. L., and Patterson, D. A., *Computer Architecture A Quantitative Approach*, San Mateo, Morgan Kaufmann, pp. 576-578, 1990.
  18. Stone, H. S., *High-Performance Computer Architecture*, Reading, Massachusetts, Addison-Wesley, pp. 24-103 and pp. 292-324, 1993.
  19. Hennessy, J. L., and Patterson, D. A., *Computer Architecture A Quantitative Approach*, San Mateo, Morgan Kaufmann, pp. 403-485, 1990.
  20. Hennessy, J. L., and Patterson, D. A., *Computer Architecture A Quantitative Approach*, San Mateo, Morgan Kaufmann, pp. 92, 167, 1990.
  21. Oppenheim, A. V., and Schaffer, R. W., *Discrete-Time Signal Processing*, Englewood Cliffs: Prentice Hall, pp. 609-618, 1989.
  22. Stone, H. S., *High-Performance Computer Architecture*, Reading, Massachusetts, Addison-Wesley, pp. 75-77, 1993.
  23. Charlesworth, A. E., and Gustafson, J. L., "Introducing Replicated VLSI to Supercomputing: the FPS-164/MAX Scientific Computer." *Computer*, Vol. 19, No. 3 pp. 10-23, March 1986.
  24. Stone, H. S., *High-Performance Computer Architecture*, Reading, Massachusetts, Addison-Wesley pp. 303, 1993.
  25. Hellerman, H., *Digital Computer System Principles*, New York: McGraw-Hill, pp. 245, 1973.
  26. Lawrie, D. H., Vora, C. R., "The Prime Memory System for Array Access," *IEEE Transactions on Computers*, Vol. C-31, pp. 435-442, May 1982.
  27. Chiueh, T., Verma, M., et al, "Efficient Implementation Techniques for Vector Memory Systems," *IEEE 1994 International Symposium on Parallel Architectures, Algorithms, and Networks*, pp. 270-277, 1994.
  28. Sohi, G. S., "High-Bandwidth Interleaved Memories for Vector Processors-A Simulation Study," *IEEE Transactions on Computers*, Vol. C-42, No. 1, pp. 34-44, January 1993.
  29. Sohi, G. S., "High-Bandwidth Interleaved Memories for Vector Processors-A Simulation Study," *IEEE Transactions on Computers*, Vol. C-42, No. 1, pp. 34-44, January 1993.
  30. Sohi, G. S., "High-Bandwidth Interleaved Memories for Vector Processors-A Simulation Study," *IEEE Transactions on Computers*, Vol. C-42, No. 1, pp. 38, January 1993.
  31. Hellerman, H., "On the Average Speed of a Multiple-Module Storage System," *IEEE Transactions on Computers*, Vol. C-15, p. 670, August 1966.
  32. Chang, D. Y., Kuck, D. J., and Lawrie, D. H., "On the Effective Bandwidth of Parallel Memories," *IEEE Transactions on Computers*, Vol. C-26, pp. 480-489, May 1977.
  33. Rau, B. R., "Program Behavior and the Performance of Interleaved Memories," *IEEE Transactions on Computers*, Vol. C-28, pp. 191-199, March 1979.
  34. Coffman, E. G., Jr., Burnett G. J., and Snowdon, R. A., "On the Performance of Interleaved Memories With Multi-Word Bandwidth," *IEEE Transactions on Computers*, Vol. C-20, pp. 1570-1573, December 1971.

- 
35. Burnett, G. J., and Coffman, E. G., Jr., "Analysis of Interleaved Memory Systems Using Blockage Buffers," *Communications of the ACM*, Vol. 18, pp. 91-95, February 1975.
  36. Dubois, M., Scheurich C., and Briggs, F., "Memory Access Buffering in Multiprocessors," *Proceedings of the 13<sup>th</sup> International Symposium on Computer Architecture*, Tokyo, Japan, pp. 434-442, June 1986.
  37. Sohi, G. S., "High-Bandwidth Interleaved Memories for Vector Processors-A Simulation Study," *IEEE Transactions on Computers*, Vol. C-42, pp. 34-44, January 1993.
  38. Baskett, F., and Smith, A. J., "Interference in Multiprocessor Computer Systems With Interleaved Memory," *Communications of ACM*, Vol. 19, pp. 327-334, June 1976.
  39. Briggs, F. A., and Davidson, E. S., "Organization of Semiconductor Memories for Parallel-Pipelined Processor," *IEEE Transactions on Computers*, Vol. C-26, pp. 162-169, February 1977.
  40. Cheung, K. C., Sohi, G. S., Saluja, K. K., and Pradham, D. K., "Design and Analysis of a Graceful Degrading Interleaved Memory System," *IEEE Transactions on Computers*, Vol. C-39, pp. 63-71, January 1990.
  41. Loomis, H. H., Jr., and Bernstein, R. F., Jr., "High Speed Pipeline Processor and Memory Architectures for Cyclostationary Processing," *Twenty Eighth Asilomar Conference on Circuits, Systems, and Computers*, Pacific Grove, California, August 1994.
  42. Bernstein, R. F., Jr., and Loomis, H. H., Jr., "Cyclostationary Processing Using Butterfly Machines," *Proceedings of the 1994 Conference on Information Science and Systems*, pp. 868-872, Princeton University, NJ, March 1994.
  43. Oppenheim, A. V., and Schafer, R. W., *Discrete-Time Signal Processing*, Englewood Cliffs: Prentice Hall, pp. 609-618, 1989.
  44. *Digital array Signal Processor a66110/a66111 User's Guide*, Array Microsystems, 1420 Quail Lake Loop, Colorado Springs, Colorado 80906, July 1991.
  45. *LH9124 Digital Signal Processor Real Time Simulator User's Guide*, Sharp Electronics Corp., Reference Code SMT90055 Rev A 4/30/93.
  46. Zimmer, M. L., "A VLSI Design of a Radix-4 Floating Point FFT Butterfly", Masters Thesis Navy Postgraduate School, December 1991.
  47. Jackson, K. L., "A CMOS, VLSI, Implementation of a FFT for Cyclic Spectral Analysis", Master's Thesis Navy Postgraduate School, March 1995.
  48. Brown, W. A., Loomis, H. H., Jr., "Digital Implementations of Spectral Correlation Analyzers," *IEEE Trans. On Signal Processing*, Vol. 8, No. 2, pp. 38-49, April 1991.
  49. Singleton, R. C., "On Computing the Fast Fourier Transform," *Communications of the ACM*, Vol. 10, pp. 647-654, 1967.
  50. Pease, M. C., "Adaptation of the Fast Fourier Transform for Parallel Processing," *Journal of the ACM*, Vol. 15, pp. 252-264, 1968.
  51. Oppenheim, A. V., and Schafer, R. W., *Discrete-Time Signal Processing*, Englewood Cliffs: Prentice Hall, pp. 599-605, 1989.
  52. Cooley, J. W., and Tukey, J. W., "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics of Computation*, Vol. 19, pp. 297-301, April 1965.

- 
53. Trivedi, K. S., *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Englewood Cliffs: Prentice Hall, 1982.
  54. Allen, A. O., *Probability, Statistics, and Queuing Theory With Computer Science Applications*, Boston: Academic Press, Inc., 1990.
  55. Hellerman, H., *Digital Computer System Principles*, New York: McGraw-Hill, pp. 244-245, 1973.



## **Appendix   *Matlab<sup>TM</sup> Source Code for STM Simulator***

```
% File Name:      stm.m
% Description:    Top level driver for split transaction memory
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:     27 Oct 95
%
% Comments:
%   3/08: Empty modified to be a SF variable rather
%         than a variable!
%   3/13: Slight cleanup of comments
%   3/16: Add event flags to catch activity to/from
%         DRAM as is done to/from CPU.
%   3/26: Modify to accept only an address. Bank #
%         is computed in gen_addr()
%   4/14: Modify to measure latency from the point of
%         of issue by the processor
%   4/22: Modify to allow both ASCII and binary output
%   4/23: Performance enhancements (init_rec)
%   10/27: Add PB bank selection
%
% function [] =
% stm(Fname,ASCII,Level,AList,NoBanks,NoCE,MemRatio,MemDecode,A)
% where
%   Fname      File name for saved data
%   ASCII      Determines the format of the output file
%   Level      Determines the level of detail of ouput saved in
%              Fname.
%   AList      Address List. This is a matrix. Each row
%              is of the form: [Address Bank# RW]
%
%   NoBanks    Number of banks to be used in the simulation
%   NoCE       Number of Cache Elements to be used in the
%              simulation
%   MemRatio   Ratio of Dynamic to Static memory cycle time
%   MemDecode
%              0 - Conventional decoding
%              1 - PB decoding using matrix A
%   A          PB decoding matrix
```

```

function [] = ...
stm(Fname,ASCII,Level,AList,NoBanks,NoCE,MemRatio,MemDecode,A)

% Check input arguments
if ((MemDecode==0) & ((nargin<8)|(nargin>9)) ),
    fprintf(1,'Input Parm Error 1\n');
    exit(-1);
end;
if ((MemDecode==1) & (nargin~=9)),
    fprintf(1,'Input Parm Error 2\n');
    exit(-1);
end;
if ((NoCE<1) | (MemDecode<0) | (MemDecode>1) | (NoBanks<1) | ...
(Level<0) | (Level>2) | (ASCII<0) | (ASCII>1) | ...
((Level==2)&(ASCII==0)) ) ,
    fprintf(1,'Input Parm Error: 3\n');
    exit(-1);
end;
if (MemDecode==1),
    ADim = size(A);
    if (2^ADim(1)~=NoBanks),
        fprintf(1,'Input Parm Error: 4\n');
        exit(-1);
    end;
    clear ADim
end;

% If Permutation based decoding is chosen, permute the addresses
% using the A matrix
if (MemDecode==1),
    Addr = AList;
    [ResultVect,NoDigits] = pb_int(Addr, A, 0);
    AList(:,1) = ResultVect';
end;

%%% Parameter initialization %%%
% Simulation Parameters
SysClk = 1;
Curlnd = 1;
%%% These variables are used for data collection %%%
MemResp = zeros(1,2); % 1st variable is Boolean
%
% 1-response occurred;
% 0-response did not occur.
% 2nd variable indicates Bank responding
ReqAllowed = zeros(1,3); % 1st variable is Boolean.
%
% 1-request was allowed;
% 0-request was not allowed.

```

```

% 2nd variable indicates Bank responding
% 3rd variable indicates address
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
LastAddr = 0;
%%% Key Parameters %%%
%NoBanks
%NoCE
%MemRatio
ReqCount = MemRatio;
%%% NoCE Adjustment for effective NoCE %%%
NoCE = NoCE + 1;
%%% Bank Variables %%%
% Note that each variable is two dimensional; the first variable is used
% to specify an element within an array (e.g., Cache variables). The
% second index is used to specify the bank number.
%
%%% Cache Array Elements %%%
Index = zeros(NoCE,NoBanks);
IndexN = Index;
Address = zeros(NoCE,NoBanks);
AddressN = Address;
RW = zeros(NoCE,NoBanks);
RWN=RW;
Ready = zeros(NoCE,NoBanks);
ReadyN=Ready;
Data = zeros(NoCE,NoBanks);
DataN=Data;

%%% Counters %%%
NAC = ones(NoBanks,1);
NACN=NAC;
CPC = ones(NoBanks,1);
CPCN=CPC;
OC = ones(NoBanks,1);
OCN=OC;
DCount = zeros(NoBanks,1);
DCountN=DCount;

%%% Flags %%%
Empty = ones(NoBanks,1);
PDC = zeros(NoBanks,1);
PDCN=PDC;

```

```

%%% Signals %%%
GRI = ones(NoBanks,1); % Initially all TRUE
GR = 0;
REI = zeros(NoBanks,1); % Initially all FALSE
RE = 0;
BS = zeros(NoBanks,1);

%%% Global Counters %%%
ReqC = zeros(NoBanks,1);
ReqCN=ReqC;
ResC = zeros(NoBanks,1);
ResCN=ResC;

ODataLen = length(AList)*2;
OData = zeros(ODataLen,9);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Program Begins Here %%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize Save File
if ASCII,
    fid=init_sf(Fname,NoBanks,NoCE,MemRatio,Level);
end;
done = 0;
fprintf(1,'Simulation Begins ... \n');
fprintf(1,'# Banks: %d\n',NoBanks);
fprintf(1,'# Cache Elements: %d\n',NoCE-1);
fprintf(1,'# Memory Ratio: %d\n',MemRatio);
fprintf(1,'# Memory References: %d\n',length(AList));
Nlog = log10(length(AList));
if (Nlog <=2),
    DelMark = 1;
    fprintf(1,'Each tic is 1 cycle\n\n');
elseif (Nlog <=3)
    DelMark = 10;
    fprintf(1,'Each tic is 10 cycles\n\n');
else
    DelMark = 100;
    fprintf(1,'Each tic is 100 cycles\n\n');
end;
Mark = 1;

```

```

while ~done,
    if (Mark >= DelMark),
        fprintf(1, '.');
        Mark = 1;
    else
        Mark = Mark + 1;
    end;
    if (rem(SysClk, 50*DelMark)==0), fprintf(1, '\n');
    end;
    GRI = eval_gri(NoBanks, NAC, OC, Empty, NoCE);
    REI = eval_rei(REI, ResC, Index, OC, Ready);
    Empty = ev_empty(NAC, OC, CPC, NoBanks);

    [Addr, BankSelNo, WRFlag, LastAddr, Curlnd] ...
        = gen_addr(Curlnd, LastAddr, AList, NoBanks, GRI);
    % Initialize recording variables for a time slice
    [MemResp ReqAllowed DRAMResp DRAMIssued] ...
        = init_rec(NoBanks, Addr);
    for BankNo = 1:NoBanks,
        %%% Respond to Memory Read %%%
        [OCN, ResCN, MemResp, OutData]= ...
            mem_resp(Index, RW, Ready, Data, NAC, CPC, ...
                OC, REI, ResC, MemResp, BankNo, NoCE, ...
                OCN, ResCN);
        %%% Service Dynamic Memory %%%
        [ReadyN, DataN, CPCN, DCountN, PDCN, DRAMResp, DRAMIssued]=
            ser_dmem(Address, RW, Ready, Data, NAC, CPC, OC, ...
                DCount, PDC, BankNo, ReqCount, NoCE, ...
                ReadyN, DataN, CPCN, DCountN, PDCN, ...
                DRAMResp, DRAMIssued);
        %%% Service Memory Request %%%
        if BankSelNo >=0,
            [IndexN, AddressN, RWN, ReadyN, DataN, NACN, ReqCN, ReqAllowed]=
                ser_memr(Index, Address, RW, Ready, Data, NAC, CPC, OC, GRI, ...
                    BS, ReqC, ReqAllowed, BankNo, Addr, BankSelNo, WRFlag, NoCE, ...
                    IndexN, AddressN, RWN, ReadyN, DataN, NACN, ReqCN);
            end; % if BankSelNo
        end; %for

    Index=IndexN; Address=AddressN; RW=RWN;
    Data=DataN; Ready=ReadyN;
    NAC=NACN; CPC=CPCN; OC=OCN; DCount=DCountN;
    PDC=PDCN; ReqC=ReqCN; ResC=ResCN;

```

```

% Evaluate the SFs in order to record the values that exist
% during the cycle. It also causes the simulation to
% complete at the correct time.
GRI = eval_gri(NoBanks,NAC,OC,Empty,NoCE);
REI = eval_rei(REI,ResC,Index,OC,Ready);
Empty = ev_empty(NAC,OC,CPC,NoBanks);
done = sim_comp(LastAddr,Empty);
% Save Results
if ASCII,
    sav_res(Index,Address,RW,Ready,Data,NAC,CPC, ...
            OC,DCount,Empty,PDC,GRI,REI,BS,ReqC,ResC,SysClk, ...
            NoBanks,BankSelNo,WRFlag,NoCE,fid,Level,MemResp,...
            ReqAllowed,DRAMResp,DRAMIssued,MemRatio);
else,
    if ReqAllowed(1);
        ADDR = Address(modulo1(NAC(ReqAllowed(2))-1,NoCE), ...
                        ReqAllowed(2));
    else
        ADDR = -1;
    end;
    if MemResp(1);
        ADDR2 = Address(modulo1(OC(MemResp(2))-1,NoCE),MemResp(2));
    else
        ADDR2 = -1;
    end;
    Epoch = [SysClk BankSelNo WRFlag ReqAllowed(1) ...
            ReqAllowed(3) ADDR MemResp(1) ADDR2 MemResp(2)];
    OData(SysClk,1:9) = Epoch;
    if ODataLen==SysClk,
        ODataLen = ODataLen*2;
        TData = OData;
        OData = zeros(ODataLen,9);
        OData(1:SysClk,1:9) = TData;
    end;
end;
SysClk = SysClk + 1;
end; %while

```

```

if ASCII,
    fclose(fid);
else
    OData = OData(1:SysClk-1,1:9);
    fname = [Fname, '.gr1'];
    fid = fopen(fname,'w');
    Tmp = [NoBanks NoCE MemRatio];
    fwrite(fid,Tmp,'long');
    fclose(fid);

    fname = [Fname, '.gr2'];
    fid = fopen(fname,'w');
    fwrite(fid,OData,'long');
    fclose(fid);
end;
fprintf(1,'\nTotal Number of Cycles= %d\n\n',SysClk-1);

```

```

% File Name:      ev_empty.m
% Description:    Evaluate the status of Empty flags
%                Internal flag within all banks.
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:      07 Mar 95
%
% function Empty = ev_empty(Empty,NAC,OC,CPC)
%
% where
%
%   Empty      Empty flag
%   NAC        Next Available Counter
%   OC         Output Counter
%   CPC        Current Processed Counter
%
function Empty = ev_empty(NAC,OC,CPC,NoBanks)

for i=1:NoBanks,
    Empty(i) = (NAC(i)==CPC(i)) & (CPC(i)==OC(i));
end;

```



```

% File Name:      eval_gr.m
% Description:    Evaluate the status of the Grant Request
%                control line based on the values of the Grant Request
%                Internal controls within each CE.
% Programmer:    Raymond F. Bernstein Jr.
% Date          Mod: 6 Feb 95
%
% function status = eval_gr(GRI)
% where
%   status      TRUE if MR active; FALSE otherwise
%   GRI         Grant Request Internal
%
function status = eval_gr(GRI)

if min(GRI)==0,
    status = 0;
else
    status = 1;
end;

```

```

% File Name:      eval_gri.m
% Description:    Evaluate the status of the Grant Request
%                Internal signal within all banks.
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:      22 Feb 95
%
% function GRI = eval_gri(GRI,NAC,OC,Empty)
% where
%   GRI          Grant Request Internal lines for the memory banks.
%               1 - indicates that bank is available;
%               0 - indicates that bank is unavailable.
%   NAC          Next Available Counter
%   OC          Output Counter
%   Empty        Empty flag
%
function GRI = eval_gri(NoBanks,NAC,OC,Empty,NoCE)

for i=1:NoBanks,
    GRI(i) = (modulo1(NAC(i)+1,NoCE)~=OC(i)) | (Empty(i)==1);
end;

```

```

% File Name:      eval_rei.m
% Description:    Evaluate the status of the Request Enable
%                Internal signal within all banks.
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:      21 Feb 95
%
% function REI = eval_rei(REI,ResC,Index,OC,Ready);
% where
%   REI          Request Enable Internal lines for the memory banks.
%               1 - indicates bank has data available;
%               0 - indicates bank doesn't have data available.
%   ResC         Response Counter
%   Index        Processing Index
%   OC           Output Counter
%   Ready        Ready flag
%
function REI = eval_rei(REI,ResC,Index,OC,Ready)

N = length(REI);
for i=1:N,
    REI(i) = (ResC(i)==Index(OC(i),i)) & Ready(OC(i),i);
end;

```

```

% File Name:      init_rec.m
% Description:    Initializes the recording variables prior to servicing
%                a bank. The recording variables, MemResp, ReqAllowed,
%                DRAMResp, and DRAMIssued are used to record
%                simulation events and are not a part of the simulation.
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      23 Jun 95  Expanded ReqAllowed to include address
%
% Evaluate GR (Grant Request)
% function [MemResp, ReqAllowed, DRAMResp, DRAMIssued] =
%                init_rec(NoBanks)
% where
%   MemResp      Two field variable used to record the memory response
%               First Field: Boolean indicating whether a memory response
%               occurred
%               Second field: Bank number of the responding field
%   ReqAllowed    Three field variable used to record whether a memory
%               request was permitted
%               First Field: Boolean indicating whether a memory request
%               occurred
%               Second field: Bank number of the responding field
%               Third Field: Memory Address
%   DRAMResp      Boolean indicating the bulk store responded in the cycle
%   DRAMIssued    Boolean indicating the bulk store was issued during the %
%               cycle
%   NoBanks       Number of banks for the memory to be simulated
%
function [MemResp, ReqAllowed, DRAMResp, DRAMIssued] = ...
                init_rec(NoBanks,Addr)

MemResp = [0 -1];
ReqAllowed = [0 -1 Addr];

DRAMResp = zeros(2,NoBanks);
DRAMResp(1,1:NoBanks) = zeros(1,NoBanks);
DRAMResp(2,1:NoBanks) = -1*ones(1,NoBanks);

DRAMIssued = zeros(3,NoBanks);
DRAMIssued(1,1:NoBanks) = zeros(1,NoBanks);
DRAMIssued(2,1:NoBanks) = -1*ones(1,NoBanks);
DRAMIssued(3,1:NoBanks) = -1*ones(1,NoBanks);

```

```

% init_sf.m
% Initialize Save File
% function fid=init_sf(fname,NoBanks,NoCE,MemRatio,Level)
%
% where
%     fid          File id of the opened save file
%     fname        File name for saved data
%     NoBanks      Number of Memory Banks in the simulation
%     NoCE         Number of Cache Elements in the simulation
%     MemRatio     Ratio of dynamic to static memory cycle
%     Level        Level of detail to save for analysis
%
function fid = init_sf(fname,NoBanks,NoCE,MemRatio,Level)

fname = [fname, '.gr'];
fid = fopen(fname,'wt');

if Level==0;
    fprintf(fid,'Number of Banks: %s ',num2str(NoBanks));
    fprintf(fid,'Number of Cache Elements: %s\n',num2str(NoCE-1));
    fprintf(fid,'Dynamic/Static Mem Cycle Time: %s\n\n',...
            num2str(MemRatio));
elseif Level==1;
    fprintf(fid,'Number of Banks: %s\n',num2str(NoBanks));
    fprintf(fid,'Number of Cache Elements: %s\n',num2str(NoCE-1));
    fprintf(fid,'Dynamic/Static Mem Cycle Time: %s\n\n',...
            num2str(MemRatio));
    fprintf(fid,'ClkBank#  WR  ReqAllowed MemResp  Bank#\n');
elseif Level==2;
    fprintf(fid,'%s\n',num2str(NoBanks));
    fprintf(fid,'%s\n',num2str(NoCE-1));
    fprintf(fid,'%s\n',num2str(MemRatio));
end;

```

```

% File Name:      mem_resp.m
% Description:    Evaluate and process a memory response if
% appropriate.
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      07 Mar 95
%
% Comments: 3/7: Empty modified to be a SF variable rather
%             than a flag.
%
% function [Index,Address,RW,Ready,Data,OC,Empty,REI,ResC]= ...
%             mem_resp(Index,Address,RW,Ready,Data,NAC,CPC, ...
%             OC,Empty,REI,RE,ResC,BankNo,NoCE)
%
% See definitions in Chapter V, Section B, Subsection 1) for definitions
%
function [OCN,ResCN,MemResp,OutData]= ...
    mem_resp(Index,RW,Ready,Data,NAC,CPC, ...
        OC,REI,ResC,MemResp,BankNo,NoCE,...
        OCN,ResCN)

OutData = -1;
RE = max(REI);
if (REI(BankNo)==1),
    OutData=Data(OC(BankNo),BankNo);
    ResCN(BankNo) = ResC(BankNo) + 1;
    OCN(BankNo) = modulo1(OC(BankNo)+1,NoCE);
    MemResp = [1 BankNo];
elseif (RE==1)
    ResCN(BankNo) = ResC(BankNo) + 1;
end;

```

```

% File Name:      modulo1.m
% Description:    Performs the remainder operation on two numbers
%                but the result is mapped to 1..K for modulo1(x,k)
% Programmer:    Raymond F. Berntsein Jr.
% Date Mod:      6 Feb 95
%
% function result = modulo1(x,k)
% where
%   k            Is the modulus number.
%   x            Is the number to be acted upon.
%
function result = modulo1(x,k)

result = rem(x,k);
if result==0,
    result = k;
elseif result<0,
    done = 0;
    while ~done,
        result = result + k;
        done = result>0;
    end; %while
end;

```

```

% File Name:      pb_int.m
% Description:    generates a bank selection sequence based on a PB
%                matrix A
%
% Programmer:     Raymond F. Berntsein Jr.
% Date Mod:      Oct 95
% Notes:
%
% function [ResultVect,NoDigits] = pb_int(Addr, A, fname)
% where:
%   ResultVect   Output vector of permuted bank numbers
%   NoDigits     Number of digits in address pattern. The return value
%   Addr         Input Address stream
%                will be the number of bits required to represent the
%                largest number in Addr.
%   A            Permutation matrix
%   fname        Name of the file to store the resulting bank selection
%                patterns.
%
function [ResultVect,NoDigits] = pb_int(Addr, A, fname)

if ((nargin~=3)),
    fprintf(1,'Invalid parameters for PB conversion type\n');
    exit(-1);
end;
% Assuming working with base 2
B = 2;
% Make it a column matrix
s = size(Addr);
if s(2)>s(1),
    Addr = Addr';
end;
% Make it addresses only (i.e., no read/write into)
if s(2)~=1,
    Addr = Addr(:,1);
end;
maxAddr = max(Addr);
Count = ceil(log10(maxAddr)/log10(2));
NoDigits = Count;
M = length(Addr);
done = 0;
i = 1;

```



```

while ~done,
    bm(:,i) = rem(Addr,2);
    Addr = fix(Addr/2);
    if i==Count,
        done = 1;
    else,
        i=i+1;
    end;
end;
bm = fliplr(bm);

% Binary version of addr (bm) is complete. Now use only the k LSBs to
% compute the bank (i.e., k is the number of columns in A
As = size(A);
bms = size(bm);
NoColA = As(2);
NoColbm = bms(2);
if (bms(2)>As(2)),
    bm = bm(:,NoColbm-NoColA+1:NoColbm);
elseif (bms(2)<As(2)),
    A = A(:,NoColA-NoColbm+1:NoColA);
end;
ResultVect = A*bm';
ResultVect = rem(ResultVect',2);

% Create a Powers matrix
PowerVect = ones(size(ResultVect));
s = size(ResultVect);
for i=0:s(2)-1,
    PowerVect(:,s(2)-i) = PowerVect(:,s(2)-i)*(B^i);
end;

ResultVect = ResultVect.*PowerVect;
ResultVect = sum(ResultVect');

if fname~=0,
    fid = fopen([fname,'.bks'],'wt');
    fprintf(fid,'%d \n',ResultVect');
    fclose(fid);
end;

```

```

% File Name:      sav_res.m
% Description:    Save the results of the pass for analysis.
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:      14 Jun 95
%
% Comments:      3/13: Provide for 2 cycles per page for full dump.
%                6/14: Add address issued to calculate all latency
%                Level 2 only
%
% sav_res(Index,Address,RW,Ready,Data,NAC,CPC, ...
%           OC,DCount,Empty,PDC,GRI,REI,BS,ReqC,ResC);
% where
%
% See definitions in Chapter V, Section B, Subsection 1) for definitions
%
function sav_res(Index,Address,RW,Ready,Data,NAC,CPC, ...
                OC,DCount,Empty,PDC,GRI,REI,BS,ReqC,ResC,SysClk,...
                NoBanks,BankSelNo,RWFlag,NoCE,fid,Level,MemResp,...
                ReqAllowed,DRAMResp,DRAMIssued,MemRatio)

if Level == 0; % Full Dump
    fprintf(fid,'***** Clk=%3d *****\n', SysClk);
for k=1:NoBanks,
    fprintf(fid,'Bank#: %3d\n',k);
    fprintf(fid,'***Cache Element Contents***\n');
    fprintf(fid,'No Index AddrRW Rdy Data\n');
    for m=1:NoCE,
        %% fprintf(1,'m= %d\n',m);
        fprintf(fid,'%4d %5d %5d %2d %3d %8d', ...
                m, Index(m,k), Address(m,k), RW(m,k), Ready(m,k), ...
                Data(m,k));
        if (ReqAllowed(1)==1) ...
            & (ReqAllowed(2)==k) ...
            & (modulo1(NAC(k)-1,NoCE)==m),
            fprintf(fid,' <--CPU\n');
        elseif (MemResp(1)==1) ...
            & (MemResp(2)==k) ...
            & (modulo1(OC(k)-1,NoCE)==m),
            fprintf(fid,' -->CPU\n');
        elseif (DRAMResp(1,k)==1) ...
            & (DRAMResp(2,k)==k) ...
            & (modulo1(CPC(k)-1,NoCE)==m),
            fprintf(fid,' <--DRAM\n');
        elseif (DRAMIssued(1,k)==1) ...
            & (DRAMIssued(2,k)==k) ...
            & (modulo1(CPC(k),NoCE)==m),

```

```

        fprintf(fid,' -->DRAM\n');
    else fprintf(fid,'\n');
    end; % if .. elseif
end; % for m=1
fprintf(fid,'NAC= %d',NAC(k));
fprintf(fid,'OC = %d',OC(k));
fprintf(fid,'CPC= %d',CPC(k));
fprintf(fid,'DCount= %d\n',DCount(k));
fprintf(fid,'Empty= %d',Empty(k));
fprintf(fid,'PDC= %d',PDC(k));
fprintf(fid,'GRI= %d',GRI(k));
fprintf(fid,'REI= %d\n',REI(k));
fprintf(fid,'ReqC= %d',ReqC(k));
fprintf(fid,'ResC= %d\n\n',ResC(k));
end; %for k=1
fprintf(fid,'ClkBank# WR MemResp Bank# ReqAllowed\n');
fprintf(fid,'%4d ',SysClk);
fprintf(fid,'%5d ',BankSelNo);
fprintf(fid,'%2d ',RWFlag);
fprintf(fid,'%7d ', MemResp(1));
fprintf(fid,'%5d ', MemResp(2));
fprintf(fid,'%10d ', ReqAllowed(1));
if ~(rem(SysClk,2)), fprintf(fid,'\n\rf');
else, fprintf(fid,'\n\n\n\n');
end;
%end; if Level==0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
elseif Level == 1; % Validate paper studies
    fprintf(fid,'%4d ',SysClk);
    fprintf(fid,'%5d ',BankSelNo);
    fprintf(fid,'%2d ',RWFlag);
    fprintf(fid,'%10d ', ReqAllowed(1));
    fprintf(fid,'%7d ', MemResp(1));
    fprintf(fid,'%5d ', MemResp(2));
    fprintf(fid,'\n');
%end; % elseif Level==1

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
elseif Level == 2;                                % Data Analysis
    fprintf(fid,'%4d ',SysClk);
    fprintf(fid,'%5d ',BankSelNo);
    fprintf(fid,'%2d ',RWFlag);
    fprintf(fid,'%10d %5d ',ReqAllowed(1),ReqAllowed(3));
    if ReqAllowed(1);
        fprintf(fid,'%4d ', ...
            Index(modulo1(NAC(ReqAllowed(2))-1,NoCE),ReqAllowed(2)));
    else fprintf(fid,' -1');
    end;
    fprintf(fid,'%7d ', MemResp(1));
    if MemResp(1);
        fprintf(fid,'%4d ', ...
            Index(modulo1(OC(MemResp(2))-1,NoCE),MemResp(2)));
    else fprintf(fid,' -1');
    end;
    fprintf(fid,'%5d ', MemResp(2));
    fprintf(fid,'\n');
end; % elseif Level==2

```

```

% File Name:      ser_dmem.m
% Description:    Service dynamic memory within a bank
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      08 Mar 95
% Comments:      3/8: modified to require ReqCount cycles to complete a
%                DRAM cycle rather than ReqCount+1 cycles.
% function [ReadyN,DataN,CPCN,DCountN,PDCN,DRAMResp,DRAMIssued]=
% ser_dmem(Address,RW,Ready,Data,NAC,CPC,OC, ...
%          DCount,PDC,BankNo,ReqCount,NoCE, ...
%          ReadyN,DataN,CPCN,DCountN,PDCN, ...
%          DRAMResp,DRAMIssued);
%
% See definitions in Chapter V, Section B, Subsection 1) for definitions
%
function [ReadyN,DataN,CPCN,DCountN,PDCN, ...
        DRAMResp,DRAMIssued]= ...
        ser_dmem(Address,RW,Ready,Data,NAC,CPC,OC, ...
                DCount,PDC,BankNo,ReqCount,NoCE, ...
                ReadyN,DataN,CPCN,DCountN,PDCN, ...
                DRAMResp,DRAMIssued)
SDRC = ~PDC(BankNo) & (CPC(BankNo) ~= NAC(BankNo)) & ...
        (RW(CPC(BankNo))==1);
SDWC = ~PDC(BankNo) & (CPC(BankNo) ~= NAC(BankNo)) & ...
        (RW(CPC(BankNo))==0);
if (SDRC==1),
    DCountN(BankNo) = 1;
    PDCN(BankNo) = 1;
    DRAMIssued(:,BankNo) = [1; BankNo; 1];
elseif (SDWC==1),
    DCountN(BankNo) = 1;
    PDCN(BankNo) = 1;
    DRAMIssued(:,BankNo) = [1; BankNo; 0];
elseif PDC(BankNo)==1,
    DCountN(BankNo) = DCount(BankNo) + 1;
    if DCountN(BankNo)==ReqCount,
        DataN(CPC(BankNo),BankNo) = 77777;
        ReadyN(CPC(BankNo),BankNo) = 1;
        CPCN(BankNo) = modulo1(CPC(BankNo)+1,NoCE);
        PDCN(BankNo) = 0;
        DRAMResp(:,BankNo) = [1; BankNo];
    end; % if
end; % elseif

```

```

% File Name:      ser_memr.m
% Description:    Service memory requests from the processor.
% Programmer:    Raymond F. Bernstein Jr.
% Date Mod:      14 Jun 95  Added address to ReqAllowed
%
% function [] ser_memr(Index,Address,RW,Ready,Data,NAC,CPC, ...
%                   OC,DCount,Empty,PDC,GRI,REI,BS,ReqC,ResC,NoCE);
%
% See definitions in Chapter V, Section B, Subsection 1) for definitions
%
function [IndexN,AddressN,RWN,ReadyN,...
        DataN,NACN,ReqCN,ReqAllowed] = ...
        ser_memr(Index,Address,RW,Ready,Data,NAC,CPC,OC,GRI,...
                BS,ReqC,ReqAllowed,BankNo,Addr,BankSelNo,RWFlag, ...
                NoCE,IndexN,AddressN,RWN,ReadyN,DataN,NACN,ReqCN)
if BankSelNo>=0,
    if (GRI(BankSelNo)==1) & (BankSelNo==BankNo),
        IndexN(NAC(BankNo),BankNo) = ReqC(BankNo);
        AddressN(NAC(BankNo),BankNo) = Addr;
        RWN(NAC(BankNo),BankNo) = RWFlag;
        DataN(NAC(BankNo),BankNo) = Addr;
        ReadyN(NAC(BankNo),BankNo) = 0;
        ReqCN(BankNo) = ReqC(BankNo) + 1;
        NACN(BankNo) = modulo1(NAC(BankNo)+1,NoCE);
        ReqAllowed = [1 BankNo Addr];
    elseif GRI(BankSelNo)==1,
        ReqCN(BankNo) = ReqC(BankNo) + 1;
    end;
end;
end;

```

```

% File Name:      ser_memr.m
% Description:    Service memory requests from the processor.
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      14 Jun 95  Added address to ReqAllowed
%
% function [] ser_memr(Index,Address,RW,Ready,Data,NAC,CPC, ...
%                  OC,DCount,Empty,PDC,GRI,REI,BS,ReqC,ResC,NoCE);
%
% See definitions in Chapter V, Section B, Subsection 1) for definitions
%
function [IndexN,AddressN,RWN,ReadyN,DataN,...
        NACN,ReqCN,ReqAllowed] = ...
    ser_memr(Index,Address,RW,Ready,Data,NAC,CPC,OC,GRI,...
        BS,ReqC,ReqAllowed,BankNo,Addr,BankSelNo,RWFlag,NoCE, ...
        IndexN,AddressN,RWN,ReadyN,DataN,NACN,ReqCN)
if BankSelNo>=0,
    if (GRI(BankSelNo)==1) & (BankSelNo==BankNo),
        IndexN(NAC(BankNo),BankNo) = ReqC(BankNo);
        AddressN(NAC(BankNo),BankNo) = Addr;
        RWN(NAC(BankNo),BankNo) = RWFlag;
        DataN(NAC(BankNo),BankNo) = Addr;
        ReadyN(NAC(BankNo),BankNo) = 0;
        ReqCN(BankNo) = ReqC(BankNo) + 1;
        NACN(BankNo) = modulo1(NAC(BankNo)+1,NoCE);
        ReqAllowed = [1 BankNo Addr];
    elseif GRI(BankSelNo)==1,
        ReqCN(BankNo) = ReqC(BankNo) + 1;
    end;
end;
end;

```

```

% File Name:      sim_comp.m
% Description:    Evaluate if the simulation is completed
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      03 Mar 95
%
% function done = sim_comp(LastAddr,Empty)
% where
%     done        1 - Indicates the the simulation is complete
%                0 - Indicates it is not complete
%     LastAddr    1 - Indicates more than one more memory references are
%                to come
%                0 - Indicates the last memory reference is being requested
%                1 - Indicates no more memory references will be requested
%     Empty       Cache Element array indicating whether a memory bank
%                is empty (i.e., no memory requests are pending to be
%                processed.
%
function done = sim_comp(LastAddr,Empty)

AllEmpty = min(Empty);
done = AllEmpty & (LastAddr<0);

```



```

% File Name:      m_anal.m
% Description:    Organize data in graphical form for analysis of memory
%                 data from stm.
% Programmer:     Raymond F. Bernstein Jr.
% Date Mod:      29 Oct 95
%
% function [TP,S,MaxL,AvgL,StdL,SSTP,TR] =
% m_anal(fname,ASCII,Apattern,WinLen,PlotFlag,Length,PrintFlag)
% where
%   fname        Name of the file containing data produced by stm
%   ASCII        Indicates whether fname is stored as ASCII or binary
%                 0 - Binary
%                 1 - ASCII
%   Apattern      Short description of the Address pattern
%   WinLen        Length of the smoothing window for throughput
%   PlotFlag      Specifies the number and types of plots
%                 0 No plot
%                 1 One plot
%   Length        Specifies # pts used in a plot
%   PrintFlag     0 - Print to Screen
%                 1 - Print to postscript file
%                 2 - Print directly to default printer
function [TP,S,MaxL,AvgL,StdL,SSTP,TR] = ...
    m_anal(fname,ASCII,Apattern,WinLen,PlotFlag,Length,PrintFlag)

if (ASCII<0)|(ASCII>1)|(WinLen<0)|(PlotFlag<0)|...
    (PlotFlag>1)|(Length<0)|(PrintFlag>2)|(PrintFlag<0),
    fprintf(1,'m_anal::Incorrect parameters!!!\n');
    exit;
end;

% Read data in from the file
if ASCII,
    fname1 = [fname, '.gr'];
    fid = fopen(fname1,'rt');
    NoBanks = fscanf(fid,'%d',1);
    NoCE = fscanf(fid,'%d',1);
    MemRatio = fscanf(fid,'%d',1);

    [Data,COUNT] = fscanf(fid,'%d',inf);
    fclose(fid);
    NoRows = COUNT/9;
    for i=1:NoRows,
        DAry(i,1:9)= Data((i-1)*9+1:(i-1)*9+9)';
    end;
end;

```

```

else
    fname1 = [fname, '.gr1'];
    fid = fopen(fname1,'r');
    Tmp = fread(fid,3,'long');
    NoBanks = Tmp(1);
    NoCE = Tmp(2);
    MemRatio = Tmp(3);
    fclose(fid);
    fname1 = [fname, '.gr2'];
    fid = fopen(fname1,'r');
    [Data, COUNT] = fread(fid,inf,'long');
    fclose(fid);
    NoRows = COUNT/9;
    for i=1:9,
        DAry(1:NoRows,i) = ...
            Data((i-1)*NoRows+1:(i-1)*NoRows+NoRows);
    end;
end;

% Calculate the Latency
% Handle the first one seperate
CAddr = DAry(1,6);
k = 1;
while DAry(k,8)~=CAddr,
    k = k+1;
end; %while
OAry(1,1) = k;
LastLatency = OAry(1,1);
% Now do the remaining rows
for i=2:NoRows,
    if DAry(i,5)==-1,
        OAry(i,1) = LastLatency;
    elseif DAry(i,5) == DAry(i-1,5),
        OAry(i,1) = LastLatency;
    else
        k=i;
        while (DAry(k,6)==-1),
            k = k + 1;
        end;
        CIndex = DAry(k,6);
        while DAry(k,8)~=CIndex,
            k = k+1;
        end; %while
        OAry(i,1) = k-i+1;
        LastLatency = OAry(i,1);
    end;
end;

```



```

%% Speed Up
S = TotalThroughPut*MemRatio;

%% Clean Up
OAry(:,2) = OAry(:,2)*0.25; % Request Allowed (GR)
OAry(:,3) = OAry(:,3)*0.50; % Memory Response (RE)

%% Graphics Plot
if Length==0 | Length>length(OAry),
    Length = length(OAry);
    XAxisLbl = 1:Length;
else
    XAxisLbl = 1:Length; % Use user specified length
end;

if (PlotFlag==0),
    % Do nothing
else % Plot one figure
    if (PrintFlag==0),
        figure;
    end;
    subplot(3,1,1);
    plot(XAxisLbl,OAry(1:Length,1));grid;
    ylabel('Latency');
    title(['Plot ID: ',Apattern, ...
        '# Banks=',num2str(NoBanks), ...
        '# CEs=',num2str(NoCE), ...
        'Mem Ratio=',num2str(MemRatio)]);
    axis([0 Length 0 MaxL*1.2]);
    subplot(3,1,2);
    plot(XAxisLbl,OAry(1:Length,4));grid;
    ylabel('Throughput');
    title(['S=',num2str(S), ...
        'Avg TP=',num2str(TotalThroughPut), ...
        'MaxL=',num2str(MaxL), ...
        'AvgL=',num2str(AvgL), ...
        'StdL=',num2str(StdL)]);
    axis([0 Length 0 1.2]);
    subplot(3,1,3);
    plot(XAxisLbl,OAry(1:Length,2:3));grid;
    xlabel('Time (Cycles)');
    ylabel('STM Status');
    title(['SSTP=',num2str(SSTP), ...
        'TR=',num2str(TR)]);

```

```

axis([0 Length 0 0.6]);
ah = gca;
set(ah,'YTick',[0; 0.25; 0.5])
set(ah,'YTickLabels',{' '; 'GR'; 'RE'});
If (PrintFlag==1),
    set(gcf,'PaperPosition',[0.25 2.5 5.8 8.2]);
    eval(['print ',fname,' -deps2']);
    title(['S=',num2str(S), ...
        'Avg TP=',num2str(TotalThroughPut), ...
        'MaxL=',num2str(MaxL), ...
        'AvgL=',num2str(AvgL), ...
        'StdL=',num2str(StdL)]);
end;
if (PrintFlag==2),
    orient tall
    print;
end;
end;
end;

```



## INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2. Library, Code 13 Naval Postgraduate School Monterey, CA 93943-5000	2
3. Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4. Dr. Randy Roberts MEE3, MS J580 Los Alamos National Laboratory Los Alamos, NM 87545	1
5. Dr. William Brown Mission Research Corporation 2300 Garden Road Monterey, CA 93940	1
6. Dr. William A. Gardner Department of Electrical and Computer Engineering University of California, Davis Davis, CA 95616	1
7. Naval Research Laboratory Attn: CAPT Dwight Dennson, Code 9110 4555 Overlook Avenue SW Washington DC 20375	1
8. Naval Research Laboratory Attn: LCDR Barbara Bell, Code 9110 4555 Overlook Avenue SW Washington DC 20375	1

- |     |  |   |
|-----|--|---|
| 9.  | Professor Maurice D. Weir, Code MA/Wc<br>Department of Mathematics<br>Naval Postgraduate School<br>Monterey, CA 93943-5216                                   | 1 |
| 10. | Professor Herschel H. Loomis, Jr., Code EC/Lm<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121   | 1 |
| 11. | Professor Charles W. Therrien, Code EC/Ti<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121       | 1 |
| 12. | Professor R. Clark Robertson, Code EC/Rc<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121        | 1 |
| 13. | Professor Richard W. Hamming, Code CS/Hg<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943-5118                           | 1 |
| 14. | Professor Douglas J. Fouts, Code EC/Fs<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121          | 1 |
| 15. | Professor Michael Shields, Code EC/SI<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121           | 1 |
| 16. | Professor Raymond F. Bernstein, Jr., Code EC/Be<br>Department of Electrical and Computer Engineering<br>Naval Postgraduate School<br>Monterey, CA 93943-5121 | 4 |